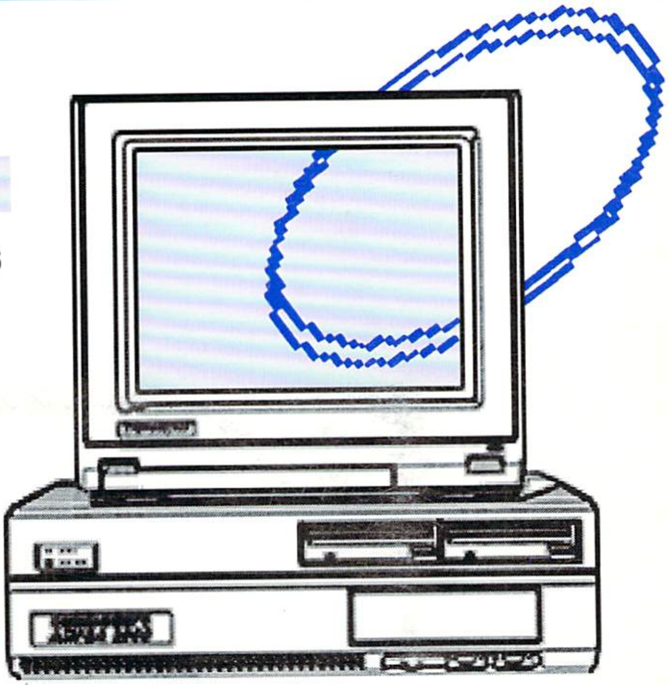
Disk Inside

Disk Inside

source and executable code for all articles

AMIGA WORLD TECH JOURNAL

INCLUDES DISK!
U.S.A. $15.95
Canada $19.95



ARTICLES

REVIEWS

COLUMNS

On Disk

(See page 25)

1.3 include files: ***a $20 value***

Enforcer: ***an MMU protection tool***

Plus source and executable code for articles

0 74470 12070 4 07

$15.95

Volume 1, Number 2 June/July 1991 An IDG Communications Publication

Captain's Log 217:2055
Time shift equalization routines recoded in
Aztec C. All systems go. It's good to be back home.

Space Staton 2055 - Viido Polikarpus

New Release
Aztec C 5.2

- **Amiga**
- **Macintosh**
- **MS-DOS**
- **80x86, 680x0 ROM**

Call Today for Special Discount Prices and Technical Information

Aztec C is available for Amiga, Macintosh, MS-DOS, Apple II, TRS-80, and CPM/80. Native and cross development.

Aztec Embedded C hosts on MS-DOS and Macintosh. Targets include: 68000 family, 8086 family, 8080/Z80/64180, & 6502

Aztec C includes - C compiler, assembler, linker, librarian, editor, source debugger, and other development tools.

MANX 800-221-0440

• Outside USA: 908-542-2121 FAX: 908-542-8386 • Visa, MC, Am Ex, C.O.D, domestic & international wire • one and two day delivery available

Manx Software Systems • 160 Avenue of the Commons • Box 55 • Shrewsbury NJ 07702

Linda Barrett Laflamme, *Editor*

Louis R. Wallace, *Technical Advisor*

Barbara Gefvert, Beth Jala, Peg LePage, *Copy Editors*

Peer Review Board

Rhett Anderson, John Foust, R.J. Mical
Gene Brawn, Jim Goodnow, Eugene Mortimore
Brad Carvey, Scott Hood, Bryce Nesbitt
Joanne Dow, David Joiner, Carolyn Scheppner
Keith Doyle, Sheldon Leemon, Leo Schwab
Andy Finkel, Dale Luck, Tony Scott
Steve Tibbett, John Toebes

Mare-Anne Jarvela, *Manager, Disk Projects*

Laura Johnson, *Designer*

Alana Korda, *Production Supervisor*

Debra A. Davies, *Typographer*

Kenneth Blakeman, *National Advertising Sales Manager*

Barbara Hoy, *Sales Representative*

Michael McGoldrick, *Sales Representative*

Giorgio Saluti, *Associate Publisher, West Coast Sales, 2421 Broadway, Suite 200, Redwood City, CA 94063 415/363-5230*

Heather Guinard, *Sales Representative/Partial Pages, 800/441-4403, 603/924-0100*

Meredith Bickford, *Advertising Coordinator*

Wendie Haines Marro, *Marketing Director*

Laura Livingston, *Marketing Coordinator*

Margot L. Swanson, *Customer Service Representative, Advertising Assistant*

Lisa LaFleur, *Business and Operations Manager*

Mary McCole, *Video Sales Representative*

Susan M. Hanshaw, *Circulation Director, 800/365-1364*

Pam Wilder, *Circulation Manager*

Lynn Lagasse, *Manufacturing Manager*

Roger J. Murphy, *President*

Bonnie Welsh-Carroll, *Director of Corporate Circulation & Planning*

Linda Ruth, *Single Copy Sales Director*

Debbie Walsh, *Newsstand Promotion Manager*

William M. Boyer, *Director of Credit Sales & Collections*

Stephen C. Robbins, *Vice-President/Group Publisher*

Douglas Barney, *Editorial Director*

The AmigaWorld Tech Journal (ISSN 1054-4631) is an independent journal not connected with Commodore Business Machines Inc. *The AmigaWorld Tech Journal* is published bimonthly by IDG Communications/Peterborough Inc., 80 Elm St., Peterborough, NH 03458. U.S. Subscription rate is $69.95, Canada and Mexico $79.95, Foreign Surface $89.95, Foreign Airmail $109.95. All prices are for one year. Prepayment is required in U.S. funds drawn on a U.S. bank. Application to mail at 2nd class postage rates is pending at Peterborough, NH and additional mailing offices. Phone: 603/924-0100. Entire contents copyright 1991 by IDG Communications/Peterborough Inc. No part of this publication may be printed or otherwise reproduced without written permission from the publisher. **Postmaster**: Send address changes to *The AmigaWorld Tech Journal*, 80 Elm St., Peterborough, NH 03458. *The AmigaWorld Tech Journal* makes every effort to assure the accuracy of articles, listings, and circuits published in the magazine. *The AmigaWorld Tech Journal* assumes no responsibility for damages due to errors or omissions. Third class mail enclosed.

Bulk Rate
US Postage Paid
IDG Communications/Peterborough Inc.

MESSAGE PORT

Time to turn pro.

ANYONE CAN BE a programmer or hardware hacker, but there's a measure of prestige in being a *registered* developer. . .not to mention inside information on the system, tech support, and hardware discounts. To step up from ardent hobbiest to cool professional, all you need to do is sharpen your powers of persuasion and pull out your check book.

Depending on your company's prosperity and progress, you may apply to be a Certified or Commercial Developer. Becoming a Certified Developer costs you $100 for the first year and $75 per year thereafter. The initial fee for Commercial Developer status is $500; renewals are $450. You need more than a check, however, to be accepted. Certified Developer applicants must prove they are seriously working on a marketable product, while potential Commercial Developers must have a product on the market (any platform will do). On both levels, Commodore gives special consideration to companies interested in developing video, graphics, educational, or business/productivity products.

Designed for companies that have small budgets and need less support, Certified Developer status entitles you to pre-release information on new system software and products, early copies of the system software, invitations to developers' conferences and seminars, discounts on Amiga hardware, technical support via BIX (plus a BIX discount), and subscriptions to the two Commodore Applications and Technical Support (CATS) newletters. *Amiga Mail* covers technical matters, while *Amiga Mail Market* discusses industry news and business issues.

Commercial Developers reap more benefits. In addition to the perks for Certified Developers, those with Commercial status receive over-the-phone technical support, larger hardware discounts, access to the closed commercial conferences on BIX, discounts on DevCons and seminars, plus they are eligible to test and review pre-release products.

For an official application, either send a self-addressed, stamped envelope to Leslie Hoopes, Administrative Representative, CATS, 1200 Wilson Dr., West Chester, PA 19380-4231, or contact her on BIX as cats.admin. Certified applications are usually processed within a week of receipt. Commercial applications can take a bit longer, as each is reviewed by the entire CATS technical staff. If it passes muster, you will be contacted and told to forward your check.

If you aren't ready to step over to the pro side, you can still benefit from the CATS pool of knowledge. Nondevelopers may subscribe to *Amiga Mail* for $45 per year and purchase the notes from past DevCons ($75 for the most recent, $20 for older volumes). Commodore also offers the *A500/A2000 Technical Reference Manual* ($40), the *Amiga 1000 Schematic and Expansion Specifications* ($20), and the *IFF Manual and Disk* ($20). Read up, and you may yet join the registered ranks. ■

[illegible]

An Introduction To the Zorro III Bus

The new expansion standard makes its mark.

By Dave Haynie

COMMODORE FIRST ENTERED the 32-bit computer market with board-level enhancements to the Amiga 2000; the A2620 arrived in 1988, and the A2630 followed in 1989. A full 32-bit system soon seemed the inevitable next step from the hybrid machines these cards created. Around the same time the 68000-based Amiga expansion capabilities, although quite advanced when introduced in 1985, were starting to show their age. So, in the summer of 1988 we at Commodore started to look into expansion capabilities beyond that of the A2000's expansion bus. The result of this effort was the expansion bus of the Amiga 3000, called the Zorro III Bus.

PERSPECTIVE AND PLANS

Commodore's original bus specification for the A1000, conceived as an external backplane cage, called for a 100-pin bus based primarily on the 68000 signals, with a few Amiga system clock and control signals added. The card itself would be not quite square, with bus fingers on one long edge and connectors on the other long edge opposite. When the bus specification was created, the Amiga prototype motherboard was called Zorro. The example backplane schematics were therefore labelled Zorro Expansion, and the Zorro Bus was born.

When the A2000 work started in 1986, the engineers and management decided to replace some of the unused and technically unusable Zorro bus signals and to change the card form factor. They chose the long rectangular shape of today's cards mainly to allow an IBM PC/AT bus connector to sit in line with an Amiga card, making bridge cards possible. While this seemed a small change to these engineers, it caused a massive shakeup in the Amiga hardware industry. Even though the effective change was mechanical not electrical, many companies could not afford a second version of their card in such a new and small market. With the birth of what the industry christened Zorro II, some of the first third-party hardware manufacturers stopped developing.

This history set the basic logical and physical constraints for the Zorro III bus. Its main goal was to provide enhanced expansion bus capability without sacrificing compatibility with the Zorro II definition. It had to have no form factor change and it had to work with properly designed Zorro II cards. I threw in an additional constraint, in light of what are considered Zorro Bus optional extensions, such as the PC/AT bus. The video slot is now a second such extension (this was planned since the West Chester engineering group took over A2000 development); allowing one-card access to both video and expansion signals. Rather than extending the Zorro II bus connector for Zorro III and limiting the number of Zorro III slots in a system, I decided Zorro III would use the same 100-pin connector as Zorro II, allowing all slots in a 32-bit system to be Zorro III. We were left with two reserved pins and two unusable pins on the Zorro II bus to work with; all others were needed.

The second goal of Zorro III, as it started out, was enhanced performance over Zorro II—something faster and more modern. After looking over the alternatives, we saw this meant primarily that Zorro III would be a full 32-bit bus. It would have a 32-bit data path to instantly double bus bandwidth, all else being equal, and a 32-bit address bus to eliminate the address space crunch already taking place in the 8.5 megabytes reserved for Zorro II cards in the A2000 and other backplanes. Other Zorro II bus features (shared interrupts and true bus masters, for example) could get improved protocols under Zorro III, as well.

We also specified a couple of new features. Most modern expansion buses are equipped to deal with block or burst transfers, where some kind of optimization can be made to transfer localized data very quickly. Because there were likely to be A3000s with two CPU clock speeds, the question of "how fast is fast" came up. Zorro II locked us into 7 MHz 68000 cycles for 16-bit designs and always will. This new bus would need to work well with faster and different processors; it should stay viable for many years. We solved the problem by designing the Zorro III protocol to be processor speed and type independent: Bus events occur based on strobes driven by the current bus master and responding slave; no bus clock is used. Plus, they favor no particular microprocessor bus design.

THE BUS CYCLES

The Zorro III bus we ended up with is a fully asynchronous, partially multiplexed 32-bit bus. All bus transactions involve three separate entities: the bus master, the bus slave, and the bus controller. The bus controller only generates a few signals, but it is responsible for managing multiple master and slave interactions. The bus master is responsible for defining most of the cycle: what kind of cycle it is, when it starts, the memory address, the data transfer direction, and what is being written during a write cycle. The slave simply responds to the bus address, accepting data supplied during writes and supplying data during reads. In the A3000 implementation of Zorro III the Fat Buster custom chip acts as both bus controller and default bus master (it converts 68030 bus protocol to and from Zorro III), but that is only an implementation detail, not a requirement.

Refer to Table 1 for the bus signal names. Bus cycles are all ►

Step Up To The Podium!

Admit it. You're an expert. You know how **it** works better than (almost) anyone. When you write code, you play by all the rules, yet your programs consistently out perform even those of the most wild-eyed ROM-jumping hacker. It's been obvious to you for some time that you should sit down at the keyboard, fire up your trusty text editor, and write an article explaining exactly how and why **it** should be done **your** way.

If the above description seems to fit you to a T, perhaps we should be talking. *The AmigaWorld Tech Journal* is looking for technical writers who have expertise in one or more areas and have a burning desire to share that information with the Amiga technical community. We need experts in all aspects of programming the Amiga, from operating systems to graphics to the Exec. You can write in any language you like—C, Assembly, Modula II, or BASIC. Best of all, you can include as much source code as you need, because all source and executable is supplied to the reader on disk. We also need hardware maestro's who can explain—in thorough detail—the inner workings of such complex components as the Amiga's chip set, expansion bus, and video slot. Don't forget algorithms either, we'll help you pass on your theories and discoveries.

The AmigaWorld Tech Journal offers you an unparalled opportunity to reach the Amiga's technical community with your ideas and code and to be *paid* for your efforts as well. So, whatever your "**it**" is that you want to write about, *The Tech Journal* is the place to publish it.

We encourage the curious to write for a complete set of authors guidelines and welcome the eager to send hardcopies of their completed articles or one-page proposals outlining the article and any accompanying code. Contact us at:

The AmigaWorld Tech Journal
80 Elm St.
Peterborough, NH 03458
603/924-0100, ext. 118

defined by the assertion of a new strobe signal, called FCS* (Full Cycle Strobe). Figure 1 shows the basic full cycle. Prior to starting a full cycle, the current bus master places the full 32-bit address on the bus, using multiplexed lines AD31. . . AD8, static lines SA7. . .SA2 and FC2. . .FC0, and READ. The LOCK* signal may also be driven to assure that the bus master does not lose the bus to another master and that shared-memory coprocessors stay out of shared memory during multicycle hardware semaphore operations. The cycle officially starts when the bus master asserts FCS*. All slave de- ►

Zorro III Signals

Pin	Name	Function
1	GND	Ground
2	GND	Ground
3	GND	Ground
4	GND	Ground
5	+5VDC	Main Supply, 2A
6	+5VDC	Main Supply, 2A
7	OWN*	Zorro II Bus Ownership
8	−5VDC	−5V @ 60mA
9	SLAVE*	Slave active strobe
10	+12VDC	+12V @ 500mA
11	CFGOUT*	Configuration chain out
12	CFGIN*	Configuration chain in
13	GND	Ground
14	C3*	3.55—3.58 MHz clock
15	CDAC	7.09—7.16 MHz clock
16	C1*	3.55—3.58 MHz clock
17	CINH*	Cache Inhibit (OVR* for Zorro II)
18	MTCR*	"Burst" Strobe (XRDY for Zorro II)
19	INT2*	Interrupt Level 2
20	−12VDC	−12V @ 60mA
21	SA5	Static Address 5
22	INT6*	Interrupt Level 6
23	SA6	Static Address 6
24	SA4	Static Address 4
25	GND	Ground
26	SA3	Static Address 3
27	SA2	Static Address 2
28	SA7	Static Address 7
29	LOCK*	Bus lock
30	AD8	Address/Data 8
31	FC0	Function Code 0
32	AD9	Address/Data 9
33	FC1	Function Code 1
34	AD10	Address/Data 10
35	FC2	Function Code 2
36	AD11	Address/Data 11
37	GND	Ground
38	AD12	Address/Data 12
39	AD13	Address/Data 13
40	INT7*	Interrupt Level 7
41	AD14	Address/Data 14
42	INT5*	Interrupt Level 5
43	AD15	Address/Data 15
44	INT4*	Interrupt Level 4
45	AD16	Address/Data 16
46	BERR*	Bus Error
47	AD17	Address/Data 17
48	MTACK*	"Burst" Acknowledge
49	GND	Ground
50	E	709—716 KHz clock
51	DS0*	Data Strobe 0
52	AD18	Address/Data 18
53	RESET*	Bidirectional Reset
54	AD19	Address/Data 19
55	HLT*	Halt
56	AD20	Address/Data 20
57	AD22	Address/Data 22
58	AD21	Address/Data 21
59	AS23	Address/Data 23
60	BR*	Bus Request
61	GND	Ground
62	BGACK*	Bus Grant Acknowledge
63	AD31	Address/Data 31
64	BG*	Bus Grant
65	AD30	Address/Data 30
66	DTACK*	Data Transfer Acknowledge
67	AD29	Address/Data 29
68	READ	Read Strobe
69	AD28	Address/Data 28
70	DS2*	Data Strobe 2
71	AD27	Address/Data 27
72	DS3*	Data Strobe 3
73	GND	Ground
74	CCS*	Compatibility Cycle Strobe
75	SD0	Static Data 0
76	AD26	Address/Data 26
77	SD1	Static Data 1
78	AD25	Address/Data 25
79	SD2	Static Data 2
80	AD24	Address/Data 24
81	SD3	Static Data 3
82	SD7	Static Data 7
83	SD4	Static Data 4
84	SD6	Static Data 6
85	GND	Ground
86	SD5	Static Data 5
87	GND	Ground
88	GND	Ground
89	GND	Ground
90	GND	Ground
91	SENSEZ3	Zorro III Backplane Sense
92	7M	7.09—7.16 MHz clock
93	DOE	Data phase enable
94	IORST*	I/O Reset
95	BCLR*	Bus Master Clear
96	INT1*	Interrupt Level 1
97	FCS*	Full Cycle Strobe
98	DS1*	Data Strobe 1
99	GND	Ground
100	GND	Ground

Become a part of the *AmigaWorld* Programming Team

We're looking for quality programs to support the growth of the *AmigaWorld* product line, and we need your help.

▼

GAMES
ANIMATION
3D
UTILITIES
CLIP ART
AMIGAVISION APPLICATIONS
OTHER STAND-ALONE APPLICATIONS

▲

We offer competitive payment and an opportunity for fame.
Send your submissions or contact us for guidelines:

Amiga Product Submissions
Mare-Anne Jarvela
(603) 924-0100
80 Elm Street
Peterborough, NH 03458

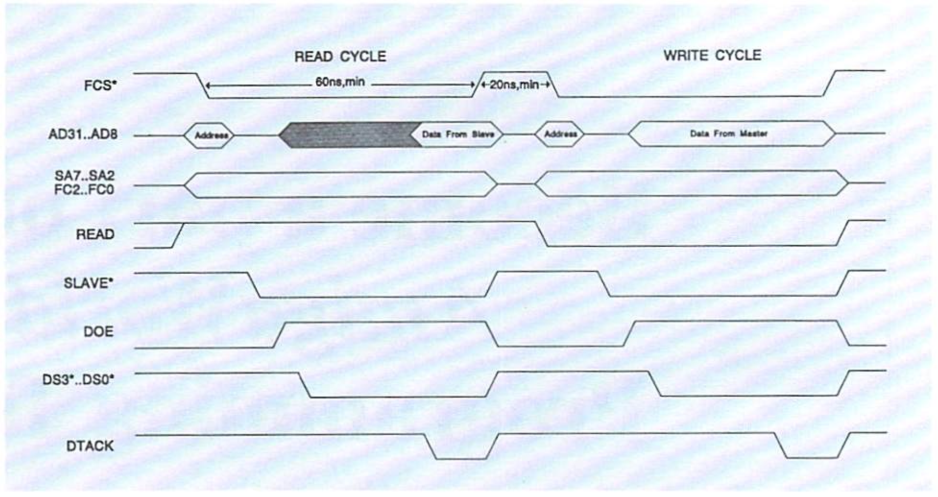


Figure 1: The basic Zorro III Full Cycle.

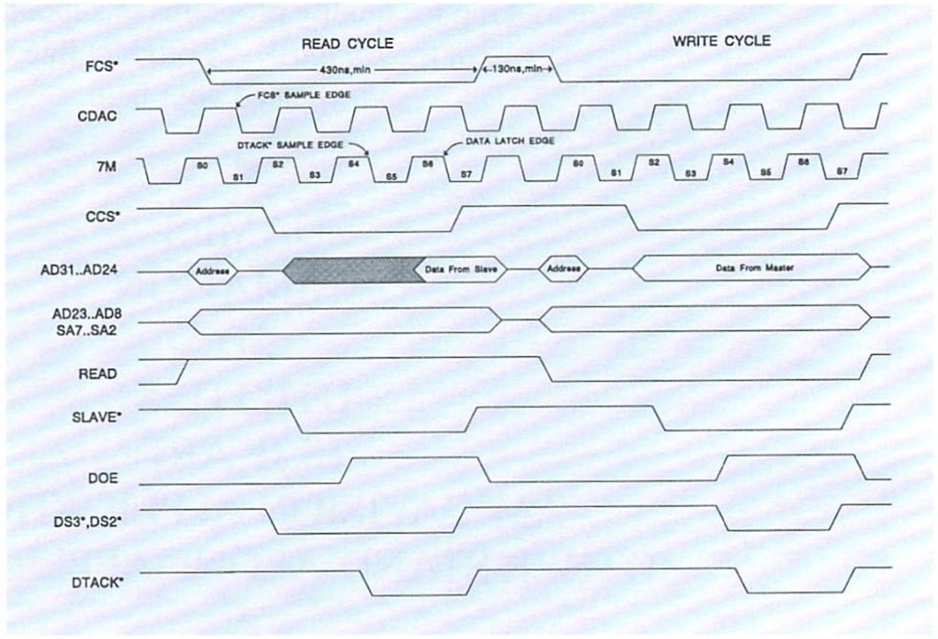


Figure 2: Zorro II as a Zorro III subcycle.

vices latch as much of AD31. . .AD8 as they require on the falling edge of FCS*. The address stays no longer than 10 ns after FCS*, then the "address phase" concludes. As in Zorro II, in Zorro III at most one card will respond to the address by asserting its private SLAVE* line. The responding slave will also assert the CINH* line if the addressed location should not be cached.

The "data phase" begins when the bus master asserts the DOE strobe, indicating that data may be driven onto bus lines AD31. . .AD8 and SD7. . .SD0. The logical to physical correspondence is not obvious here: Bus wires AD31. . . AD24, SD7. . .SD0, AD23. . .AD16, and AD15. . .AD8 correspond to logical data bits D31. . .D24, D23. . .D16, D15. . .D8, and D7. . .D0, respectively. Shortly after DOE, one or more of the data strobe lines, DS3*. . .DS0*, is asserted to indicate which of the four bytes are actually of interest and, possibly, to latch data on write cycles. Cachable slaves will ignore these strobes on reads, returning instead the entire addressed longword. As soon as the slave device has write data latched or has read data valid on the data bus it will assert the DTACK* strobe. This tells the bus master that it can latch read data and end the cycle. The cycle ends when the bus master removes FCS* and other signals from the bus, which prompts the slave to removes its signals as well.

ZORRO II CYCLES

A Zorro II Cycle is a special case of a Full Cycle, as shown in Figure 2. The cycle starts with FCS* asserted by the bus master. If the address on the bus is in the Zorro II memory range, a Zorro II subcycle is generated. FCS* will be sampled by the CDAC clock's falling edge, and when this sampling logic finds FCS* low, the Compatibility Cycle Strobe, CCS* (which corresponds to the 68000-bus AS* signal), is driven on the next rising edge of the 7M clock. The high-order address lines AD31. . .AD24 are tristated shortly after FCS* is asserted, but AD23. . .AD8 and SA7. . .SA2 are maintained throughout the cycle, corresponding directly to the standard Zorro II 24-bit address lines. The Zorro III LOCK* signal conveys the low-order address signal A1, while DS3* and DS2* correspond to the Zorro II UDS* and LDS* signals. The CINH* line becomes the Zorro II OVR* line, and the MTCR* line becomes the Zorro II XRDY signal. Zorro II did not have any cache support, so the conventions adopted for 68030 coprocessor cards are applied: The eight-megabyte Zorro II re-

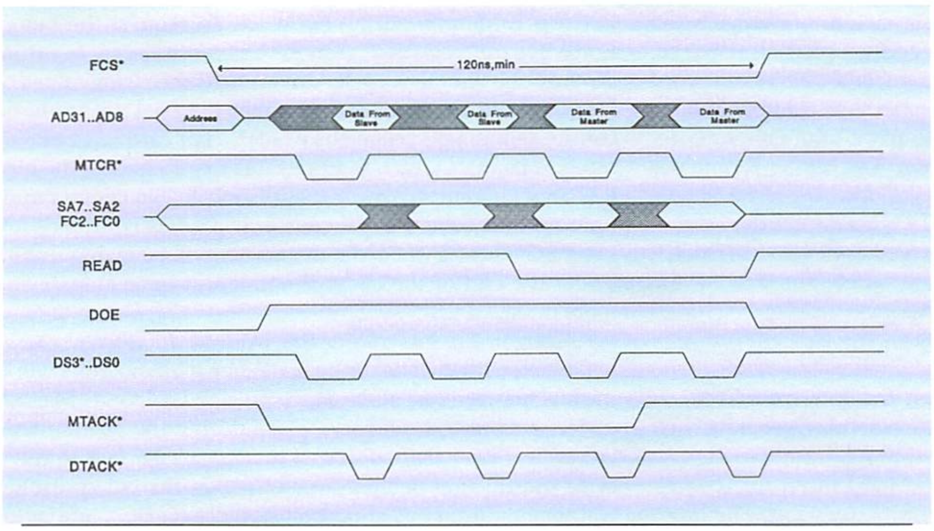


Figure 3: A Full Cycle with Multiple Transfer subcycles.

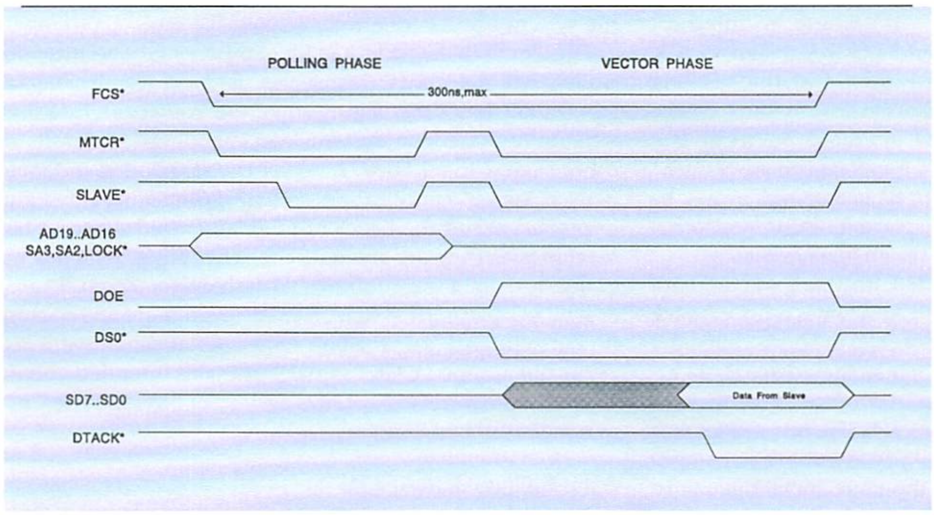


Figure 4: The Interrupt cycle.

gion is cachable; all the rest is not. The 16 data lines D15. . .D0 are carried by AD31. . .AD24 and SD7. . .SD0, respectively, which accounts for the unusual data mapping in 32-bit cycles. As on any Zorro II bus, DTACK* will be automatically generated by the bus controller, unless otherwise specified by the bus slave via OVR*.

When the default bus master (the 68030 for the A3000) accesses the Zorro III bus, the access usually will be for either Zorro II or Zorro III address space. Because of this and, to some extent, the implementation details of the A3000 bus controller, Zorro II cycles generated by the bus controller are always inscribed within Zorro III cycles. When an alternate bus master is running things, however, they are probably not. Certainly, Zorro II bus masters in a Zorro III system do not generate any Zorro III "wrapper." Instead, the strobe that starts the cycle (FCS* or CCS*) determines the cycle's type. Most Zorro III bus masters will probably not run Zorro II cycles—not because they cannot, but because the extra complexity will not be used in the majority of designs.

There is a bit more to the Zorro II support than simply mapping Zorro II signals over those of Zorro III. An unfortunate fact of life is that a considerable number of Zorro II cards on the market were somewhat sloppy in meeting the specified bus-timing conventions. (To be fair, the Commodore documentation was far from perfect.) The Zorro III bus controller snoops the Zorro II slave signals and will refuse to start a new cycle until the current one is actually terminated, regardless of when it was supposed to finish. If it did not, an errant Zorro II card addressed in one cycle could drastically interfere with a much faster Zorro III card addressed in the following cycle.

MULTIPLE TRANSFER CYCLES

Another Zorro III subcycle is the Multiple Transfer Cycle, more easily referred to as the "burst" cycle. This mechanism, illustrated in Figure 3, permits multiple 32-bit data transfers to take place within a single full cycle, much faster than ordinary full cycles can run. A burst cycle starts out as a plain Zorro III cycle, with addresses on the bus and FCS* asserted by the bus master. When a burst-capable slave responds to the bus address, in addition to asserting its private SLAVE* line, it also asserts the Multiple Transfer Acknowledge strobe (MTACK*). In the data phase, if the bus master is capable of handling the burst, it will assert the Multiple Transfer Cycle Request strobe (MTCR*). This first cycle is normally terminated by the slave with DTACK*. That DTACK* will not, however, terminate the full cycle. Instead, FCS* stays asserted and the bus controller negates MTCR*. It next supplies new values for the static address lines A7. . .A2 and possibly the READ strobe. A new subcycle begins with MTCR* and one or more of the DS3*. . .DS0* lines. The effective address is based on the latched high-order and new ►

low-order address lines. Because slaves cannot change based on the low-order address and no address phase is required, this so-called "short cycle" can transfer a 32-bit data word much faster than a single full cycle can. Any number of short cycles can take place and will take place until either master or slave can no longer handle them. The master terminates a burst by negating FCS* with MTCR* after the last subcycle. Each time the master drives MTCR*, it samples the state of the MTACK*. When it finds the slave has negated MTACK*, the subcycle defined by that MTCR* will be the last one; the full cycle will terminate with DTACK*.

INTERRUPT CYCLES

The final Zorro III cycle type is the Interrupt Cycle, as shown in Figure 4. Interrupting devices may generate their own interrupt vector, rather than sharing the automatic vectors used by Zorro II and motherboard interrupt sources. This makes servicing an interrupt extremely fast. The Interrupt Cycle starts with all function-code lines FC2. . .FC0 high, the value 1111 on AD19. . .AD16, and an interrupt number from 7. . .0 on A3, A2, and LOCK* (taken as A1). FCS* and MTCR* are driven together, and the "polling phase" begins. Unlike other types of cycles, any slave device on the bus that has asserted the indicated interrupt and needs to supply a vector for it will assert its private SLAVE* line. As quickly as 30 ns after MTCR* is asserted, it may be negated again. If no slave has responded, FCS* is also negated and the interrupt serviced via an autovector.

If at least one slave has responded, however, MTCR* and DS0* are asserted, and one of the SLAVE* lines is driven back by the bus controller to a responding slave. This is called the "vector phase." That selected slave may then drive its eight-bit vector number onto data lines AD15. . .AD8 of the bus and terminate the cycle with DTACK*. The bus master will then negate MTCR*, DS0*, and FCS*. Nothing is complicated about this cycle; speed is the issue more than complexity. The whole interrupt cycle must be kept very short, so that autovectored interrupts, which the system defaults to in the absence of a poll-phase response, are not unduly delayed. The time from FCS*, asserted by the bus master, to DTACK*, asserted by the slave, is never longer than 200 ns.

BUS ARBITRATION

The Zorro II bus had a perfectly functional bus-acquisition mechanism, essentially just an arbitrated extension to the basic 68000 three-wire bus-takeover protocol. There are, however, some problems with this design. First, it has no concept of fairness. A very bus-hungry card in the first slot will always win the bus in lieu of any card further upstream, regardless of the activity of the other cards. In fact, two extremely hungry cards can easily starve a third for extended periods, plus they can starve the host CPU, causing interrupt responses to be overly long. It is also left up to a bus-master's design to be friendly and not hog the bus; once a card has the bus, the card keeps it until volunteering to give it up. If the card runs into trouble, it may hang the system. Additionally, there is no simple way for a bus master to fluidly share the bus, other than via constant request, master, and relinquish cycles, which waste time in arbitration better spent doing real work.

Zorro III uses a new bus-acquisition protocol. Instead of the three/four-wire interface of Zorro II, Zorro III uses only BR* and BG*; the bus controller manages BGACK* and OWN* for bus-buffer direction control and for the benefit of Zorro II cards monitoring BGACK*. To request the bus, a Zorro III card "registers" with the bus controller by asserting BR* for one 7M clock cycle, driven between rising edges of 7M. This tells the bus controller that the card wishes to be considered as a potential bus master. That card idles until it receives its BG*, which indicates it is free to master the bus. It is only guaranteed one bus cycle, which can contain multiple transfer subcycles, of course. If the bus controller negates BG* during the cycle, the master will be giving up the bus at the end of the cycle; if not, it will have at least one more cycle.

As long as the bus master has work to do, it may remain registered with the bus controller. When it has no more work, the bus master signs off by again pulsing BR* for a 7M clock cycle. The pulse should take place during the bus master's last cycle, rather than after it. If the bus master fails to run a cycle shortly after being granted the bus, the bus controller will time out that bus-allocation slot, negate BG*, and automatically sign off the card. This prevents cards from tying up the system because of error, or idling the bus.

The Zorro III method has several benefits. The bus controller, not the cards themselves, is now responsible for scheduling bus time. This allows the controller to adjust scheduling based on system load, pending interrupts, or other events. In addition, it makes the scheduling independent of the bus specification, so that it can be modified and improved to suit the needs of more advanced Zorro III bus machines as they are created. The scheduling, which is actually done in a fair round-robin fashion for Zorro II cards as well in a Zorro III system, is made even more fair because no Zorro III card hogs the bus. A bus master can prevent loss of the bus via the LOCK* signal, but that is intended only to support hardware semaphores and other atomic cycles, not as a means of bus-hogging.

AUTOCONFIGURATION

One of the more innovative features of Zorro II was the AUTOCONFIG™ protocol, which let the operating system manage the placement of expansion cards and the automated matching of hardware and support software. Zorro III continues and extends this capability. Physically, the same daisy-chain mechanism is used for configuring each board in turn. Now, however, there are two places to locate the AUTOCONFIG registers. They may appear at Zorro II configuration base \$00E8xxxx, where they are physically 16-bit registers and must obey all Zorro II rules. This is attractive for cards that use the SENSEZ3* line to support both Zorro II and III protocols. Alternately, the registers may be asserted at the new configuration base, \$FF00xxxx, as 32-bit Zorro III bus registers. In either case, all read-registers are physically present on the high-order nybble of the bus, either logically D15. . .D12 if Zorro II-mapped or D31. . .D28 if Zorro III-mapped. All AUTOCONFIG registers are logically eight bits wide, made up of nybble pairs. Figure 5 shows the logical-to-physical mapping of Zorro III registers for each configuration base. The only difference in addressing is the exchange of A1 in 16-bit mode for A8 in 32-bit mode. All AUTOCONFIG registers are named by the address offset from the base of their physical high nybble. For example, register 04, the product number, has its high nybble at location BASE+4 for either configuration space and its low nybble at BASE+6 if the Zorro II base, BASE+\$104, is the alternate base.

Only a few AUTOCONFIG registers are changed in Zorro

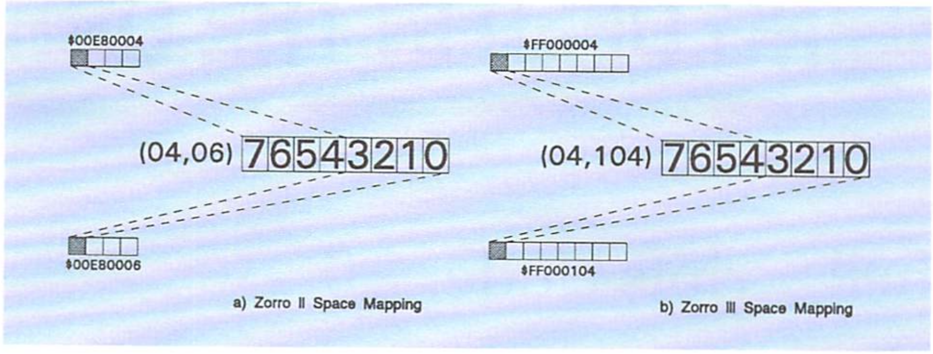


Figure 5: Physical-to-logical register mapping.

III. Bits 7 and 6 of register 00 now read "10", rather than "11", to indicate a Zorro III card. Bits 2. . .0 of that same register indicate the old Zorro II configuration sizes or alternately extended sizes between 16MB and 1GB, depending on the size-extension bit, bit 5 of register 08. Bit 7 of 08 now indicates whether the board is basically a memory device or an I/O device. Bit 4 of register 08 is fixed at 1, which corrects a bug in Amiga OS 1.3 that would sometimes erroneously identify a Zorro III board as a Zorro II board. Bits 3. . .0 are perhaps the most interesting new feature. Under Zorro II, a card with free memory had to be completely filled with memory to be automatically linked by the operating system. This usually resulted in memory boards with jumper settings to adjust to different memory populations. We at CBM don't like Amiga boards to have jumpers, if they can be avoided. So, bits 3. . .0 allow a board to specify a logical subsize of memory within the physical configuration size. A board may have matching physical and logical sizes, or it may specify several logical sizes between 64K and 14MB. Or, much like with chip and fast memory in the A3000, the operating system can automatically determine how much memory is installed. All other read-registers are the same as in Zorro II.

As for write configuration, the normal "shut up" feature is maintained at register 4C. A Zorro III board using the Zorro II configuration base is configured, typically, by writing the A31. . .A24 byte to register 44, then the A23. . .A16 byte to register 48. For the new configuration base, A31. . .A16 is typically written as a word to register 44, configuring the board. There are some esoteric nybble and byte writes corresponding to these byte and word writes, respectively, but they are seldom used in expansion cards.

TOWARD THE FUTURE

While there are far more details about Zorro III than I have recounted here (the specification I wrote for Commodore is about 86 pages long), I have covered the main points of the new bus. For more information, timing, and mechanical specifications, consult "The Zorro III Bus Specification," which is available from Commodore Applications and Technical Support. As background reading on the Zorro II bus, I recommend *The A500/A2000 Technical Reference Manual*, which is available from CATS, as well. Essential to the understanding and design of Zorro II cards is *The 68000 User's Manual*, available from Motorola. (See "The Off-Line Library" for addresses.)

It is hard to tell now if Zorro III will be as popular as Zorro II: Fast designs are harder and more expensive to build than slow designs, and Amiga 3000s (and presumably any future Zorro III machines) at the moment are less in need of expansion devices than A2000s were. What you can put in an A3000 slot is a bit more interesting than the typical A2000 fare (memory and a hard disk), which is covered for the majority of users on the A3000 motherboard.

Still, the Zorro III bus seems to have achieved the design goals set for it; nearly all the Zorro II cards we tested in it worked (and so far flaws in the card, not Zorro III, have accounted for those that did not). The bus has proven to run nearly as fast as the A3000 motherboard bus, which is no small accomplishment for an architecture as clock- and processor-neutral as this. Zorro III should provide a home for fast peripherals, video boards, processing engines, and hard-disk-sized memory boards for years to come. ■

Dave Haynie has been with Commodore since 1983 and was Senior Design Engineer on the A2000, A2620, A2630, and A3000 projects. Write to him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or contact him on BIX (hazy).*

AAMIGA WAREHOUSE

The Memory Specialist

"FOR SOFTWARE GO TO THE REST. FOR HARDWARE CALL THE BEST!!!"

A Power User's Lunch
Memory, Accelerators, Flicker Fixer's.
I almost forgot dessert, Hard Drives.
You can get your fill without emptying your wallet !

Do you do any Desktop Publishing or Ray Tracing? Using Amax? Then you know how anoying Interlace flicker can be. We have packages that can save your eyes .

Call for current pricing on Memory chips. We will not be Beat on any product we have in stock.

Special
Flicker Fixer and VGA Monitor
$575

Hard Cards Start as low as $353.50 for a 40 meg system

We carry a full line of Monitors.

Limited Time
CSA's Mega Miget with CPU and Co Proccessor for only $699.50

GOLDEN*IMAGE*
ICD
Expansion Systems

15488 Feldspar Dr.
Chino Hill, CA. 91709
1-800-942-9505
714-283-0499
Major Credit Cards accepted or Cod.

1-800-942-9505

Shell Scripts Under 2.0

By Andy Finkel

SCRIPT WRITING INVOLVES warping innocent programs into doing things that they were not really designed to do. To make this less tortuous, Amiga OS 2.0 offers a number of new features in the dos.library, Shell, and AmigaDOS commands for some interesting results. (This article is based on the 2.04 version of the operating system. Because of bug fixes, certain elements may work slightly differently than in earlier versions.) For more flexibility, the Shell now supports local variables, variable expansion in prompts and the command line, and backticks, which let you use the output of one command in another.

CHANGEABLE CHANGES

Under 2.0, local variables are local to the Shell that created them. You specify and clear them with the SET and UNSET commands. While each Shell has its own copy of the variables, they are copied to child processes. Changes a child process makes, however, do not affect the parent's copies of the variables.

If your program requires wider reaching variables, use global environment variables. Global variables are similar to local variables, except that there is only one instance of each variable, which is stored in ENV:. You set and clear global variables with the SETENV and UNSETENV commands. Any change to a global variable is all inclusive: All Shells see the change.

In addition, the Shell now expands variables (both local and global) it encounters in prompts and the command line. When you preface a variable name with $, the Shell searches the local variables and then the environment variables. If it finds a match, it replaces the variable name with the proper value. For example:

```
SET name andy
ECHO "My name is $name."
```

produces the output:

```
My name is andy.
```

To prevent variable expansion, preface the $ with the escape character *.

```
ECHO "My name is *$name."
```

outputs

```
My name is $name.
```

The Shell automatically sets three useful local variables for you: RC is the return code of the previous command, Result2 is the secondary result of the previous command, and Process is the current process number. In addition, the VERSION command in the Startup-sequence automatically sets the Kickstart and Workbench variables, which are useful in scripts for version checking.

Finally, debugging your scripts is easier under 2.0, thanks to the echo variable. If you set echo to on, as in:

```
SET echo on
```

the system will print each command to the screen before executing it. This is extremely helpful in tracking a script's control flow, showing you what is happening, step by step, and which line causes the script to exit.

MULTIPLE ARGUMENTS

Most of the AmigaDOS commands now take patterns and multiple arguments. Both are useful directly on the command line, but multiple arguments are a boon in scripts, especially when combined with another new feature, backticks (' symbols). Enclosing a command in backticks in a prompt or command line, tells the system to insert the output of the command directly in the command line. Any returns enclosed are changed to spaces. This allows you to feed the output of one command into the command line of another. For example:

```
TYPE 'LIST #?.info lformat="%P%N"' hex
```

types all the .info files from the current directory in hex. (The example also uses the 2.0 ability of most AmigaDOS commands to accept multiple arguments.)

You can even use backticks in aliases:

```
ALIAS vers VERSION *'which [ ]*'
```

This handy alias gives you a version command that searches paths. Note the use of the * escape character to prevent the backticks from being executed when the alias was typed in.

Finally, a reminder: The question mark (?) has two special abilities when used in scripts. The ? prompts for *missing* arguments only, and it can redirect the input prompt. Why is this so powerful? Suppose you want to copy a user-specified file to df1:Tools in a script. The easy solution is:

```
ECHO "What file please ?" NOLINE
COPY >NIL: to df1:tools ?
```

That's all there is to it. The output redirection to NIL: sends the template to NIL: to make the script output cleaner.

Redirecting the input stream for a command combined with the use of ? lets you extract command line arguments from files. Local variables, Shell variable expansion, and the backtick have eliminated the need for this technique for the most part, but it can still be useful in some situations. For example:

```
date >ram:qwe
setdate <ram:qwe >NIL: filename ?
```

could be phrased as:

```
set fdate
setdate filename $fdate
```

or:

```
setdate filename 'date'
```

under 2.0.

ASK FOR REDIRECTIONS

You probably noticed that under 2.04, the Shell supports redirection anywhere on the command line. (To revert to 1.3 behavior, just set the local variable $oldredirect to on.) Now, the combination < > redirects both input and output to the same interactive file handle (which is useful with CON:). For example:

```
DIR < >CON:////SPECIAL inter
```

performs an interactive dir in its own window. The >> (append) redirection symbol now creates a file if one does not exist. Plus it opens the file in shared write-access mode, allowing other programs to inspect the file as it is being created.

Changing their addresses instead of their features, many common script commands now reside in ROM. These newly built-in commands are: ALIAS, ASK, CD, ECHO, ELSE, ENDCLI, ENDIF, ENDSHELL, ENDSKIP, FAILAT, FAULT, GET, GETENV, IF, LAB NEWCLI, NEWSHELL, PATH, PROMPT, QUIT, RESIDENT, RUN, SET, SETENV, SKIP, STACK, UNALIAS, UNSET, UNSETENV, and WHY. You can use them with no speed penalty and without risking a disk swap. If you prefer a customized version of any of the commands, you can make it resident using the add keyword. This would give it priority over the built-in version. Alternatively, the RESIDENT command's remove option disables the built-in version. In this case, the system searches the normal command paths for your command. This may be necessary if your replacement command is not pure and cannot be placed on the resident list.

Similarly, stubs for .key, .bra., and .ket have been placed on the resident list, letting you put these commands into any script. Note: Unless you are executing a script (or starting it from the Shell with its script bit set), the commands will do nothing. They are there to prevent scripts from breaking when used as a from option in NEWSHELL/NEWCLI.

NEW LOOKS ON OLD FACES

Under 2.0, the LIST command is a much more powerful script generator. Not only does it support the all keyword, allowing you to create recursive scripts, but it also has greatly expanded lformat options. Take a look:

%A	Print file Attributes (protection bits).
%B	Print size of file in Blocks.
%C	Print file Comment.
%D	Print file Date.
%K	Print file Key block.
%L	Print Length of file in bytes.
%N	Print Name of file.
%P	Print file parent Path.
%T	Print file Time.

You can even put a length or justification specifier or both between the % symbol and the field specifier. Don't worry, the 1.3 %S behavior is still supported. LIST lformat is especially useful in combination with the backticks.

ECHO also has two new options for 2.0. First, it takes multiple arguments, allowing you to leave off the quotes if you prefer.

```
ECHO This is a test
```

produces

```
This is a test
```

(Note that each of the strings is separate, and is operated on separately by the first and len keywords.) Because of this, you can use ECHO to read and process the first line in a file for a script. In addition, the ECHO command has a new to option that lets you use ECHO to parse command lines or pick up input without fussing with the command template. (Remember that * can be used for output to the current standard output.)

The SEARCH command now has useful returns, so you can branch off the results of a search. It also has some handy optional output modes.

GENERAL RULES

Always put in .bra { and .ket } statements, whether you intend to use script variables or not. (The Shell will ignore those statements if you do not use script variables.) The important thing is to remove the confusion between the dual use of < > for both redirection and script variables (an unfortunate choice that we are still forced to live with). While the 2.0 EXECUTE command does better at figuring out which you really mean when you say:

```
<hi >there
```

an element of confusion still exists. You are better off explicitly setting the script variable delimiters from the start.

The 2.0 VERSION command can pick up version strings from commands, libraries, devices, file systems, and scripts. You may want to take advantage of this by including a version string in your script. The format is:

```
$VER: name VERSION #.REVISION # ddd-mm-yy
```

Finally, remember to set the script bit on your scripts, both AmigaDOS and ARexx. The Shell can tell the difference between AmigaDOS and ARexx scripts, and will invoke the appropriate interpreter.

To see some of these new features in action, consult the Finkel directory on the accompanying disk. The scripts provided should give you a start on writing your own. Remember, the kludgier you get, the more interesting and powerful your scripts can be. If the task becomes too complex, however, I advise writing it in ARexx. ■

Andy Finkel is Manager of Amiga Software at Commodore and head of the 2.0 project. He's been with the company since before the VIC-20 and is one of the people to thank for approving the purchase of the Amiga. Contact him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or on BIX (afinkel).*

HAM Algorithms

By Jamie Purdon

HOLD-AND-MODIFY (HAM) graphics mode uses a hardware trick to display all the Amiga's possible 4096 colors at once. HAM's main advantage is its ability to display 12-bit color (four bits for each of the red, green, and blue components). Because the Amiga can read only six bits per pixel from a bitmap, however, HAM presents some problems in terms of color encoding. The resulting color-resolution loss becomes apparent as color "fringing." You can use a number of software techniques to minimize this effect, and should take several issues into consideration when designing programs that incorporate HAM graphics.

ALL IN THE MODIFIER

HAM mode is unusual in the way it interprets pixel color. With other display modes, each pixel is assigned a value. This value serves as an index into a color table that holds the palette values. Indeed, that is how HAM works for pixel values between 0 and 15. Pixel values higher than 15 are interpreted differently, however.

Normally, HAM mode is used with six bitplanes. Each pixel can have a value between 0 and 63. If the pixel value is 0–15, then the pixel's displayed color is a palette color. If the pixel's value exceeds 15, its called a HAM modifier. In this case, the new pixel's color is the same color as the pixel to its left, except that the red, green, or blue component is modified. (For the leftmost column of pixels, the "pixel to the left" is considered to be palette color 0.) The changed color component is set to a value of 0–15 (depending on the low nybble, or four bits, of the pixel's value). In other words, you can change all three color components of a pixel by setting it to a palette color, or change just one component using a HAM modifier.

In the case of five-bitplane HAM, pixel values from 16 to 31 indicate that the blue component of the color is to be changed. In this variety of HAM, there is no way to modify the red or green components of the color. You either change the blue component or change the pixel to an entirely new palette color. While this may seem restricting, one less bitplane affects DMA in such a way as to allow more chip-bus time for the CPU, Blitter chip, and so on.

In HAM, unless a pixel matches one of the palette colors, its color is based on the pixel to its left. This can cause color artifacts that look like horizontal bleeds of red, green, or blue. For example, green bleeding will occur if a horizontal line contains only red and blue modifiers, and a green modifier is plotted. If green6 is plotted, then the green color gun will be set at value 6 for all pixels to the right of the plotted one, unless another green modifier or a palette color intervenes.

Hardware sprites do not affect the HAM pixel colors, and there is no way to query the hardware directly to determine what color is being displayed by any pixel. Also, there are no system routines that help with this sort of work. Thus, the task of computing display colors is left to applications software. This is usually done either by saving the colors that are plotted or by querying the bitmap.

The Atari Lynx uses a system, which you can duplicate in HAM, whereby any given (RGB) color component changes only in every third column of pixels. While this cuts the effective horizontal resolution by a factor of three, it allows you to quickly get going in HAM mode and has the benefit of automatically reducing color bleeding. With this method, all bleeds extend for two pixels only, since every color component is modified every three pixels.

There is a mode called 4096+ that uses 16 palette colors along with HAM modifiers (instead of using just the modifiers). This allows for better color resolution: There is less bleeding, so images look sharper. Because a palette color changes all three RGB color components, you can clean up color bleeds much more quickly by using palette colors rather than using only HAM modifiers.

Another mode, which NewTek refers to as Dynamic HAM, uses the Copper to change palette colors on a scan-line by scan-line basis. This leads to more color control, sharper images, and less color fringing. Choosing which palette colors to use for such a display can get extremely complicated, however.

You can actually take advantage of HAM fringing to quickly draw horizontal lines. Consider a screen filled with the HAM modifier red0. You can create a blue or green line (actually, a color bleed) on any scan line just by plotting two pixels to determine its start and end points. You might plot a green8 at the starting pixel and then plot green0 at the ending position plus one. You can easily extend this process to draw checkerboards and more complicated graphics.

COLOR PLOTTING

Designing a method to choose a modified six-bit color to plot is probably the hardest task you face in programming for HAM mode. A related algorithm is needed to handle HAM artifacts, which are the result of color bleeding or fringing produced by HAM's inability to always match the desired 12-bit color. The two functions, determining what to plot and handling color bleeding, go hand-in-hand and are usually addressed together. Both entail using subroutines that reference the 12-bit color value of any given pixel—important information when considering which HAM modifiers to plot.

There are at least a couple of methods for determining a color in the display. One involves setting up either one or

"Determining the 'best' color is highly subjective and depends on your algorithm."

three arrays to hold the red, green, and blue values for each pixel. Arrays provide for quick access to RGB values (via table lookup), but are expensive in terms of memory.

Another method for determining display colors is to decode them on the fly. This is done by calls (often repeated) to ReadPixel( ), scanning in a leftward direction. If the plotted pixel is a palette color, only one ReadPixel( ) call is needed because the value matches a palette entry. When the plotted pixel is a HAM modifier, however, it becomes more complicated: The ReadPixel( ) call is repeated until each red, green, and blue component is known. A component is "known" when a HAM modifier for it, or a palette color, is found in the bitmap. Backward scanning stops when a palette color is reached (remember that a palette color changes all three components) or when the process reaches the left edge of the screen.

Scan-line methods are used not only for color determination but also for HAM calculation. This generally involves three main loops: decoding the current colors, color calculations (blending, and so on), and re-encoding the colors. HAM bitmaps are decoded to RGB arrays, which are then modified and re-encoded into HAM bitmaps.

Scan-line methods are effective for quickly processing many pixels. They also offer excellent speed improvements on 68020 and 68030 CPUs because the loops take advantage of these processors' code caches.

There are a number of ways to choose a six-bit HAM color. Beware that the display can look poor when you use only HAM modifiers. It improves with the use of "close enough" palette colors. If you want a row of pixels all one color, it is best to use a palette color for the first pixel, and then plot the remaining pixels with HAM modifiers that tweak the red, green, and blue values to achieve the 12-bit color desired.

You can use a pixel-plotting routine to determine which HAM modifier or palette color is best to use. Determining the "best" color is highly subjective and depends on your algorithm. Some algorithms do not use a palette color if it does not exactly match the 12-bit color to be plotted. Better (visual) results, however, are obtained by using "close enough" palette colors. The 3-D cube concept can be used to determine how closely two colors match one another. Imagine a cube where the three dimensions are represented by red, green, and blue. First, this process finds the differences in the red, green, and blue components between two colors. Then the "closeness" is calculated as the sum of the squares of the differences (this formula derives from the Pythagorean theorem):

$$(\text{red diff.})^2 + (\text{green diff.})^2 + (\text{blue diff.})^2$$

Other color systems, such as the HSV (hue, saturation, and value) and YIQ (the NTSC color signal), could be used. These methods, however, require an extra set of conversions from RGB color space to and from the other color space. After conversion to HSV, for example, the sum of the squares of the distances between "new" characteristics is used. Although code for RGB to (and from) "other color space" conversions can be highly optimized, a speed penalty is still involved.

The best palette color is the one that is closest, in 3-D space, to the desired 12-bit color. Finding the best HAM modifier (red, green, or blue) is easy: It is the one with the greatest color "distance."

Once your algorithm has determined the closest palette color and the best HAM modifier, it must select one to use. One way to do this is to compare the "closeness" of the two results. The closeness equations for HAM modifiers are basically the same as for palettes. Notice, however, that one of the terms can always be factored out, because one will always be zero. The equations are:

$$\text{red closeness} = (\text{green diff.})^2 + (\text{blue diff.})^2$$
$$\text{green closeness} = (\text{red diff.})^2 + (\text{blue diff.})^2$$
$$\text{blue closeness} = (\text{red diff.})^2 + (\text{green diff})^2$$

These algorithms are complex and execute slowly. You can segregate the code by puttting it in a subroutine, however, and use table-lookup methods to speed the process.

THE CLEAN-UP CREW

Whenever you change the color of a HAM screen pixel, you run the risk of causing color bleeds. You can, however, perform a clean-up operation to correct any unwanted color changes. This procedure moves from left to right on the screen, starting just after the original plotted pixel. It can stop when the current pixel's 12-bit color is the same as it was before. A shortcut is to stop the cleanup when an existing palette color is reached.

Here is a simple way to perform the cleanup: Before plotting your primary pixel, determine the colors of the four pixels to its right. After plotting, recompute what needs to be plotted for the four clean-up pixels to make them the same as their original 12-bit color values.

The reason that four clean-up pixels are used instead of three is that if the first is plotted with a "close enough" palette color, up to three HAM modifiers may be required to correct the (newly generated/modified color) bleed. This is rather empirical (not well grounded in theory) but works well in practice.

One way to design general-purpose HAM-plotting rou- ►

tines is to begin with a low-level function such as:

```
PlotPixel(Window,x-pos,y-pos,red,green,blue)
```

This function is not as simple as it might seem. Not only does it have to figure out the best six-bit value to plot, but it must also take care of the clean-up work. (Other functions, such as Circle( ) can be built from textbook examples.) Including the clean-up function, the PlotPixel( ) algorithm breaks down roughly as:

```
determine 12-bit color of pixel at (x−1,y)
determine 12-bit color of pixels at (x+1..4,y)
plot new color at pixel(x,y)
plot up to four clean-up pixels at (x+1..4,y)
```

The Amiga graphics.library's shape-drawing "primitives" (ellipses, rectangles, and so on) are nearly useless for dealing directly with HAM, because of the clean-up consideration. While there is no problem using only palette colors, the HAM advantage of 4096 colors is lost with this limitation. To make use of the built-in routines, you can use a single-bitplane bitmap/rastport of the same size and shape as the HAM bitmap to draw shapes with the graphics.library. You can then scan this mask to determine which pixels to color or modify in the HAM bitmap.

ALL IN A DITHER

Dithering offers a way to represent more bits per color than is directly accessible from the hardware. It trades spatial resolution for increased apparent color resolution. For example, you can represent green4.5 by plotting a pattern of alternate green4 and green5 pixels. While each pixel is exactly green4 or green5, when they are viewed over a large area, the eye perceives them as green4.5. Because multiple pixels are required to represent this in-between color, spatial resolution is compromised.

There are two general classes of dither: pixel-based and nonpixel-based. Pixel-based methods do not rely on neighboring pixels and therefore compute faster. A common pixel-based dither algorithm is the ordered dither, whereby a repeating two-dimensional pattern of fractions is added to the plot data before bits are truncated for display. A variant is the random dither algorithm, in which a random fraction is added to the plot data before truncation.

Nonpixel-based methods include the classic Floyd-Steinberg, whereby "dither errors" are propagated to neighboring pixels. A typical spatial dither first plots a pixel and then computes the error as the difference between what was desired and what was actually plotted. It then distributes $^3/_8$ of the error to the neighboring pixel on the right, another $^3/_8$ to the pixel on the next line, and the remaining $^2/_8$ to the pixel on the next line and to the right. You could design your own PlotPixel( ) function that includes ten-bit numbers for the red, green, and blue parameters. While only four bit colors will be plotted, the other six bits per color will determine the dither. Nonpixel-based dither types tend to produce wavey line patterns. They are nice for image-processing work, but tend to be computationally expensive for general-purpose pixel work.

THE BLITTER END

When you use the Blitter in HAM mode, the display may appear to flash more than you expect. This is a side effect of the CPU and Blitter being unable to keep pace with the display. The most obvious technique for minimizing this flashing is double-buffering. Sometimes, however, you may not have the chip memory to spare for this method. In such cases, you can curb the flashing by breaking each blit into a number of horizontal strips and thus limiting the flashing to these strips.

In general, the blitter is very inefficient for single-pixel work. Often, a hand-coded write-pixel routine that uses the CPU will outperform a blitter-based routine when dealing with HAM. As with any CPU-based routine, you should take care not to intrude on the rest of the system. A pixel should not be set where a menu is being displayed; the pixel will be reset when the menu disappears. Remember, Intuition menu displays appear asynchronously in application programs. An easy way around this is to avoid using menus and requesters with HAM screens.

If you do use HAM and Intuition together, you should know of another potential problem. Because palette colors become green modifiers (the top bits set in the six-bit display pixel) when highlighted, visual problems can occur with gadgets and menus. Gadgets that use highlighting tend to show ugly green artifacts, so if you use HAM imagery for gadgets, it is best to use the alternate-image facility instead of the complement mode. This problem tends to wreck the HAM display where menus overlap, and usually causes quite a bit of horizontal bleeding. One way around this is to use a hi-res (not HAM) screen for menus and gadgets. Intuition's active-window facilities allow you to keep track of whether any of an application's screens are in use.

HAM is an ingenious solution to graphics-display limitations, but, because of the way it works, programming with HAM requires special considerations. Knowing what these considerations are, however, you can make your HAM application fast, memory efficient, and flexible.

ON DISK

The on-disk example (in the Purdon drawer) is a little graphics hack, created with the Metacomco assembler, that draws designs in any of the 4096 colors around the current cursor position. The shape and color of the designs are based on the mouse coordinates. The program uses RGB arrays to store the current plotted colors, and a subroutine that uses the algorithm described above to determine which six-bit value to plot.

Except for the "Determine what to plot" subroutine, I have tried to keep the program fairly simple. This subroutine is in a separate source file, and you can add some "glue code" to make it useful for any language. Its source contains an additional subroutine called CreateDetermine, which creates a lookup table to speed processing. This lookup table contains an entry for every one of the 4096 possible 12-bit colors. Each entry contains the closest palette color and its "closeness" (distance) as determined by the 3-D method. ■

Jamie Purdon is the author of NewTek's DigiPaint and ToasterPaint software. He's made a career out of the Amiga, (quietly) programming in 68000 assembly language for the past 4½ years. Prior to that he played with Color Computers and Forth, and also wore a suit and tie for various "mini-careers": American Greetings (Prime Computer), software support (Qantel computer applications) and PeeCees (Lotus 123 consulting). Contact him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or on BIX (jamiep).*

On Disk

Building a 3-D Object Viewer

Brush up on your vector math and you're ready to start.

By Brian Wagner

PROGRAMMING A 3-D graphics viewer may seem complex, but all you need is a basic understanding of vectors and geometry plus some general math skills. The trick is how you combine the three.

As with any program, the best way to start a 3-D object viewer is to walk through the goals and concepts. With the program (we'll call it 3DView), we want to display a user-specified object with user-specified viewing parameters. In addition, the screen size should be adjustable to take advantage of the Amiga's flexible graphics capabilities.

OBJECT OF ATTENTION

The first consideration is how to represent three-dimensional objects. For simplicity, we will use the ASCII file format for 3-D objects, known as GEO. First introduced by VideoScape 3-D, GEO is now supported by a wide variety of Amiga 3-D graphics programs. Because it is an ASCII file format, you can create and edit the objects with any text editor or word processor. (For a full description of the file format, refer to the ReadMe file in the Wagner drawer of the accompanying disk).

Once we have an object, we must describe how to view it. Typically in 3-D graphics, the camera analogy is used: The camera points at the object and the picture appears on the computer equivalent of the film, the screen. In our program we will simulate a camera to describe how to view an object. More precisely, we will use a small file that contains the following data:

```
Camera position
Where the camera is looking
Viewing scale
Light source position
```

Just as an ASCII file is the simplest way to represent objects, an ASCII file works fine for our viewing description. The information must be in numeric form, however, such as:

```
0 0 50
0 0 0
1.0
50 50 50
```

Before we go any further, you should understand that all positions in 3-D space are described using the 3-D coordinate system, which is usually shown as in Figure 1. Each of the three lines (or axes) of the graph represents a dimension (or direction) in three-dimensional space. Values on these axes ►

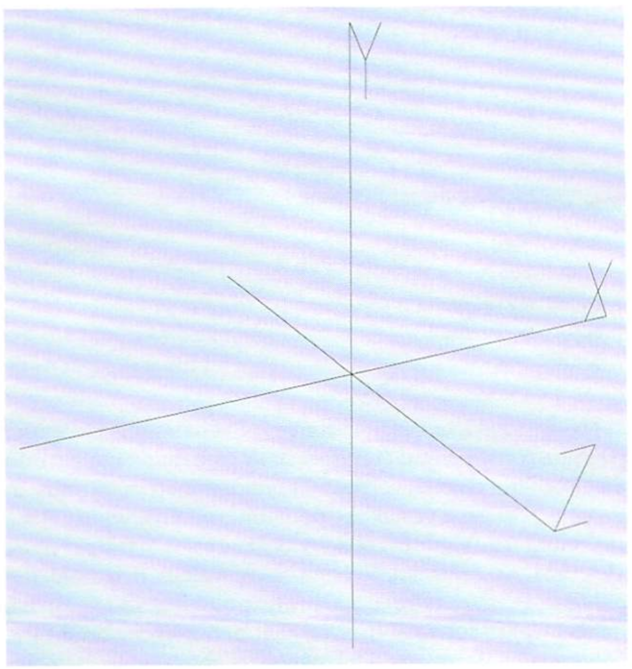


Figure 1: The 3-D coordinate system.

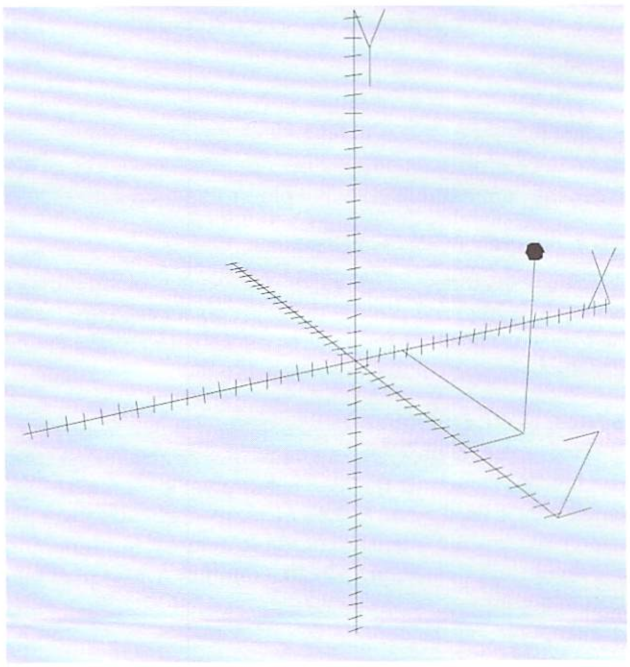


Figure 2: A position marked in 3-D space.

increase in one direction, decrease in the other, and are zero at the shared origin (center point). Just as you represent a position on a two-dimensional graph with a pair of numbers, you represent a position in 3-D with a triplet of numbers. Each number describes where on each axis the "position" is located. (See Figure 2.)

Study the viewing-description file. As you can see, the camera's position is represented by three numbers describing its location in the 3-D coordinate system. The next three numbers describe the point at which the camera is "looking." If you drew a line from the camera's position to the point the camera is looking at, you would know the "direction" in which the camera is aimed. This direction is known as a vector (more on this coming up). The next number in the file is used to scale the view of the object either smaller or larger (which we'll discuss later). The last set of three numbers describes the position of the light source that will provide the illumination. Keep in mind that the object's description also uses similar triplets describing positions in 3-D space.

One last bit of background is in order: the way objects are represented in the 3-D world. Basically, objects are composed of a number of polygons, each of which you can think of as pieces of the object's surface (see Figure 3). In turn, each polygon is made up of a number of vertices, or corners. By connecting each of a polygon's vertices, you draw the polygon they describe (see Figure 4). As described earlier, these polygon vertices are defined as positions in 3-D space via coordinates. Using this method, we can arbitrarily define polygons (and objects) anywhere in 3-D space.

We have a way of representing the object and the camera viewing it, now we must design the 3DView program that will use all this information to create a picture of the object. The program will run from the CLI or Shell, and the user will pass it the filenames of both the object and viewing-description file in the form:

```
3DView objectfile viewfile
```

Let's add one more argument that describes the display resolution. To cut down on typing, we'll use the following code numbers to represent the screen resolutions:

1	320 × 200
2	320 × 400
3	640 × 200
4	640 × 400

Now the command line is:

```
3DView objectfile viewfile screenmode
```

To be really fancy, let's say that if the screen-mode number is negative, the program will draw the object in a wire-frame display. If it's positive, the program will create a solid, shaded (by the light source) display.

STRUCTURE DEFINITIONS

With our goals clear, take a look at the program's first component, 3DView.h in the Wagner drawer's source.lzh file. (Note that I used version 3.6a of Manx C for the example code.) 3DView.h contains all of the structure definitions and a few constant definitions. The first definition is for the Polygon structure:

```
struct Polygon {
   SHORT cnt;
   SHORT *vtx;
   SHORT col;
   FLOAT nx, ny, nz;
   FLOAT cx, cy, cz;
   SHORT sha;
};
```

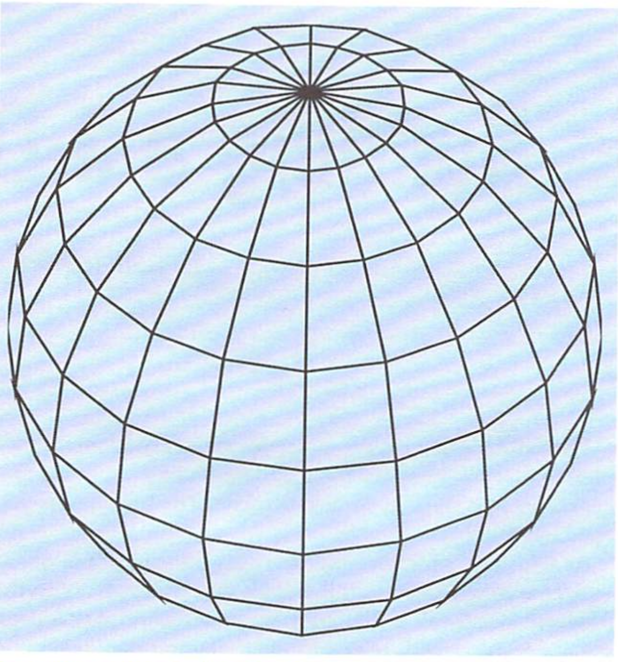


Figure 3: Many polygons make up a 3-D object's surface.

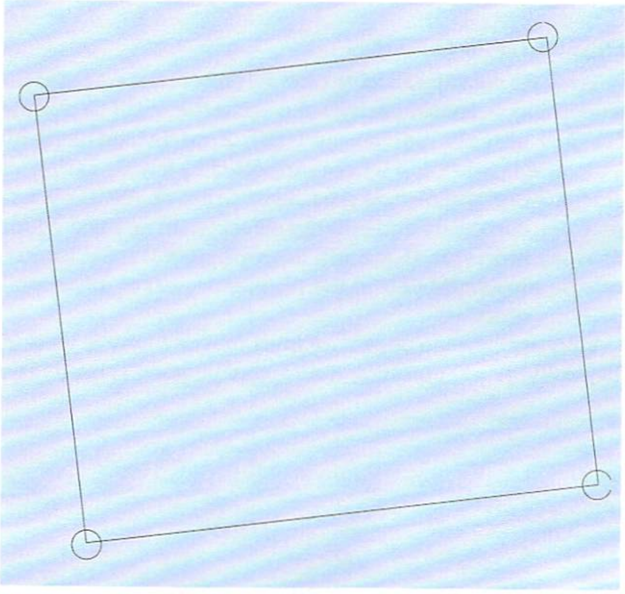


Figure 4: Each polygon is defined by vertices.

The first field, cnt, is the number of vertices this particular polygon contains, and *vtx is an array containing indices into the list of the object's vertices. Therefore, there is an entry in the array for each vertex the polygon contains (which is the cnt value). The value in col describes the color of the polygon. The next six fields are floating-point values that hold the polygon's surface normal and center point. The last field, sha, holds a value between 0 and 7 that defines how bright the polygon should appear. This, of course, is related to the polygon's orientation to the light source. I'll go into detail on the floating point values and sha a little later.

The next definition is for the Vertex structure:

```
struct Vertex {
   FLOAT x, y, z;
};
```

Vertex contains the XYZ coordinates of the vertex's location in 3-D space.

Next is the Projection structure:

```
struct Projection {
   LONG x, y;
};
```

This structure holds the X and Y coordinates of the actual points the system draws on the screen. We will end up converting each of the object's vertices into XY coordinates to draw. The Projection structure, therefore, is closely related to the Vertex structure. You could say that the Vertex structure defines points in 3-D space, while the Projection structure defines points in 2-D (screen) space.

The last structure is ViewOpts:

```
struct ViewOpts {
   FLOAT cax, cay, caz;
   FLOAT lpx, lpy, lpz;
   FLOAT scl;
   FLOAT lsx, lsy, lsz;
};
```

ViewOpts holds the same values that were specified in the viewing-description file earlier. The first three fields are the camera's location, the next three are the point at which the camera is looking. The following field is the view-scaling factor, and the final three are the location of the light source.

READY, SET, LOAD

Now, let's get into the code. The first file of interest, 3DView.c, contains only the main( ) function that does most of the Amiga-specific setup. First, it checks if the program arguments are valid and opens the graphics and intuition libraries. Next, it allocates the buffers that will hold the object, simply allocating arrays of the same structures that were defined in 3DView.h. Polygon, Vertex, and Projection each receive a separate buffer. Because we are allocating a fixed number of each structure, the program can display only objects that do not have more than the set number of polygons or vertices. The limits are defined in 3DView.h.

Following buffer allocation, the program opens the screen and window. Remember, because we are allowing different screen modes, we have to open the screen based on the specified mode. After the window is opened, more buffers are allocated for polygon drawing. These buffers are specifically required by the Amiga to do any filled-polygon drawing.

To complete the setup procedure, we define the screen palette, which the program uses for both object coloring and object shading. The first eight colors in the palette are black (background color), purple, blue, yellow, red, brown, green, and orange. The last eight colors are all grays, ranging from dark to light. We will actually mix these grays with the other colors to produce the illusion of different shades, or intensities, of each color!

With all of the buffers and graphics are ready, let's load the object. First, we set the global polygon, vertex, and projection counters to zero. Now, to load the object, the program calls the loadobject( ) function (located in the load.c source file):

```
err = loadobject(argv[1]);
```

Notice that we pass the object's filename, which was first supplied as an argument to main( ). In the function, we first check if the file is a valid GEO file by examining the first four characters. Next, the function reads in the number of vertices contained in the object, followed by the vertex list itself. Note that the vertices are read straight into the buffer that was previously allocated. The routine now reads polygons, one at a time, until it reaches the end of the file. The polygons are actually stored in the file in a fashion very similar to the way we will store them. In the file, each polygon looks something like:

```
4 0 1 2 3 13
```

The first value is the number of vertices the polygon contains, in this case 4. Because this polygon has four vertices, the next four values are the vertices used by the polygon. Actually, they point into the vertex list that was just loaded. In this example, the first four vertices in the list (0, 1, 2, 3) are used to define this polygon. The last number is simply a color code for the polygon. Notice that we call the function convertcol( ) before storing the color code. Because the color codes in the GEO format do not correspond directly to the palette we defined, we must convert the GEO color codes into values that can be used with our palette.

After the object is loaded, we call the loadvopts( ) function to load the viewing-description file:

```
err = loadvopts(argv[2]);
```

Again, we send the filename that was specified for the viewing-description file. This function simply loads all of the values from the viewing-description file and stores them in the global structure vopts.

ON DISPLAY

With the object loaded, we can start the display process. The next six functions each perform a separate job in the display process. (Notice we change the background color before each function is called; this reassures the user with feedback during the display process.) The first function called is:

```
transform( );
```

Aptly named, this function transforms all of the object's ver- ►

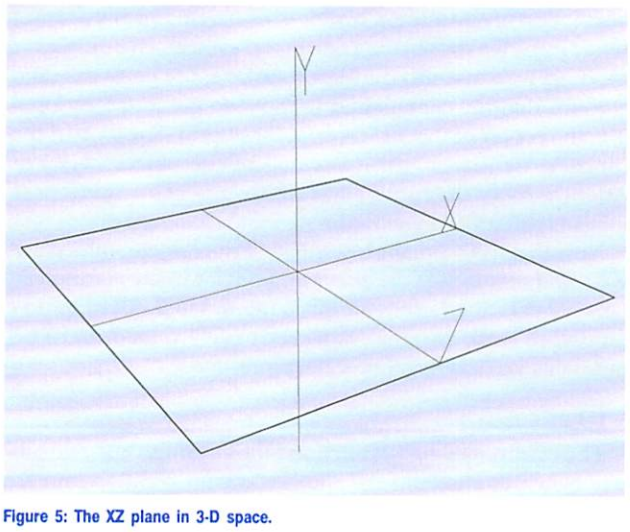


Figure 5: The XZ plane in 3-D space.

tices and the light-source position to the viewplane coordinate system. Remember from the camera analogy that your Amiga's screen simulates the camera's film. To make any further calculations much simpler, we need an easy way to represent the screen in 3-D space. The simplest is to pick one of the major 3-D "planes" to represent the screen. A plane, in 3-D terms, is a flat slice through 3-D space. The three major planes in the 3-D coordinate system are described by combining each of the three major axes in pairs, as Figure 5 illustrates. Of course, planes need not be aligned so perfectly as they are in Figure 5. In fact, they could lie anywhere in 3-D space. The easiest way to define a plane in 3-D space is to specify its "normal." Normals are vectors that are perpendicular to the plane being defined, as shown in Figure 6. For simplicity, let's choose the major XY plane to represent the Amiga's screen. The normal for the XY plane is, conveniently enough, the Z axis itself (see Figure 7).

This next part gets a little tricky, so pay close attention. For the camera, we specified its location and the point at which it is to look. We can make a vector out of these two positions by subtracting the camera position from the "lookpoint," which is the purpose of the first few lines of code in transform( ). To make vector operations simpler, the routine turns the vector into a "unit vector" by calling the function unitvector( ). A unit vector is simply a vector with a length of one and is used to define direction only. Dealing with unit vectors can greatly simplify certain functions, as you will soon see.

We now have a vector that defines not only the direction in which the camera is looking, but also the camera's "film plane" (the plane everything the camera sees will be projected onto). We want this plane to be the XY plane, however, not some arbitrary plane in 3-D space. We need to rotate the camera's film plane into the XY plane. We can do this by using the camera's direction vector and the XY plane's vector (which is the Z axis). Take a look (nx, ny, nz is the camera's direction vector):

```
s=sqrt(ny * ny+nz * nz);
if (s>0.0) {
   sx=-ny / s;
   cx=-nz / s;
}
sy=-nx;
cy=s;
```

This little bit of code from transform( ) calculates the amount of rotation around the X and Y axes needed to transform a point into the viewplane coordinate system. In addition to rotating, we must also translate (or move) the point the same amount that the camera had to move to get it to location 0, 0, 0, the center of the 3-D coordinate system. The loop in this function translates and rotates all of the object's vertices. After the loop, the routine transforms the light-source position to the viewplane coordinate system.

Now that the XY plane is simulating the camera's film plane we could almost just use the X and Y coordinates of the vertices for drawing (as one simplification). We do need to do a little more work than that, however, to determine X and Y coordinates for drawing. Enter calcprojections( ).

3 TO 2

The calcprojections( ) function loops through all of the object's vertices and converts the XYZ vertex position into XY screen points suitable for drawing. The first line in the function calculates the aspect ratio:

```
ar=(scrh * 4.0) / (scrw / 3.0);
```

The global variables scrw and scrh hold the screen's width and height in pixels. We need this scaling value because pixels are not perfectly square. (NTSC monitors are a little bit wider than they are tall.) Using the scaling value guarantees that the proper dimensions are maintained as the object goes from 3-D space to the screen.

Inside the loop, the X and Y coordinates are divided by the Z coordinate. Because we are using the XY plane as the screen, the Z coordinate tells us how far from the plane (or screen) the point lies. By dividing the X and Y coordinates by this "distance" value, we can simulate the effect of polygons in the distance appearing smaller than those that are closer to the XY plane (screen). If the Z coordinate is zero

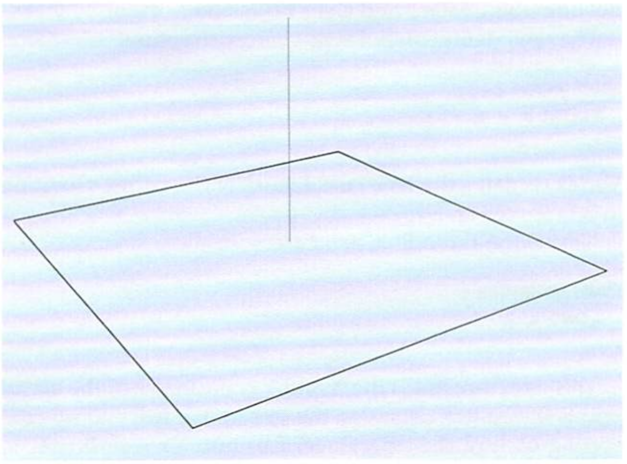


Figure 6: A plane and its normal.

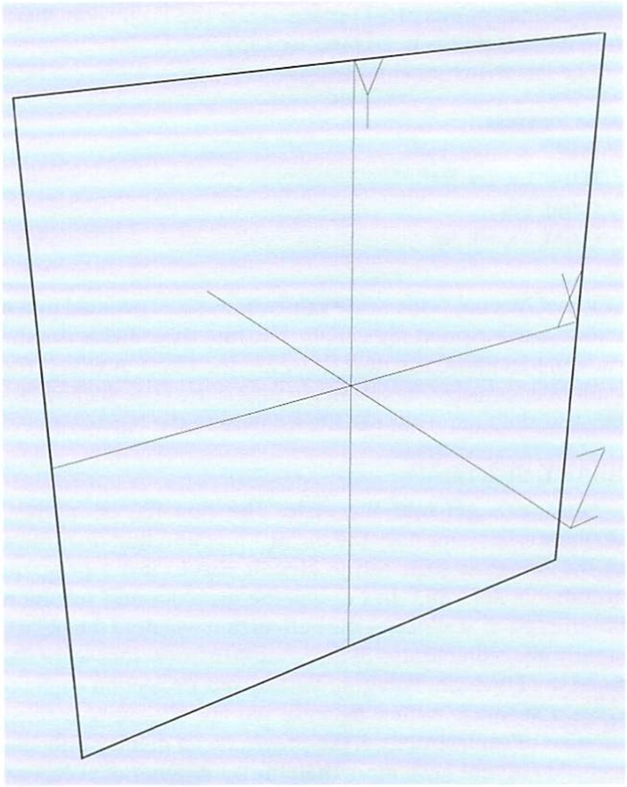


Figure 7: The Z axis is the normal of the XY plane.

(meaning the point is right on the screen), we just ensure that the XY coordinates are so huge they will not be visible. After the divisions, we scale the new X and Y values by the viewing description scaling value. This gives the user control over the scale of the projections that will be drawn, and is necessary because we do not know the scale at which the object is defined in the GEO file. If the size of the vertex coordinates is large, a small view-scaling value will be needed. If, on the other hand, small vertex coordinates are used, larger view-scaling values may be needed.

Next, we use that aspect-ratio scaling value: We scale the new Y coordinate by the scaling value. The following two lines finish the projection and store the results in the projection array:

```
projs[i].x=(tx * scrw)+(scrw / 2);
projs[i].y=(ty * scrw)+(scrh / 2);
```

First, we multiply the new XY coordinates by the screen width to convert the values, whatever they may be, to screen pixel coordinates. Why multiply the Y coordinate by the screen's width instead of its height? Because the screen's dimensions are not square (width and height aren't equal), we just say that they are and let the aspect ratio take care of the difference. Remember, we scaled the Y coordinate by the aspect ratio right before this multiplication. This lets us simulate that we are displaying on a perfectly square screen on a perfectly square monitor. The additions at the end of each line simply align the coordinates so that the center of 3-D space equates to the center of the screen. Without this adjustment, the center of 3-D space appears in the upper-left corner of the screen—where 0, 0 is located on the screen.

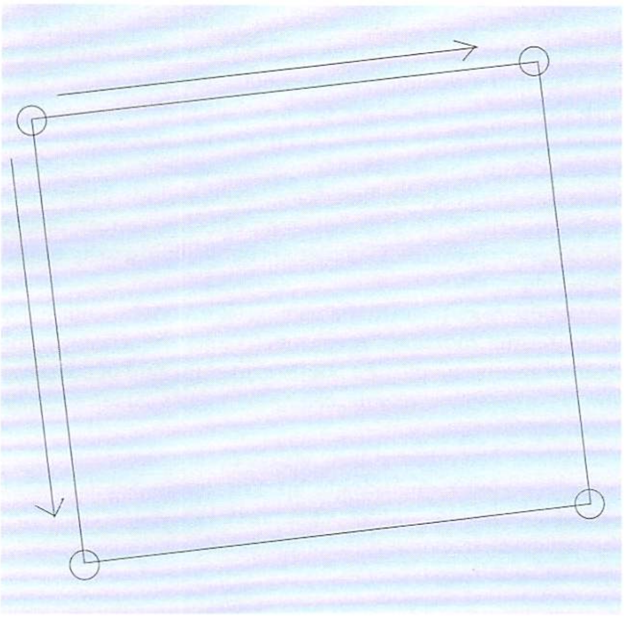


Figure 8: From the two vectors that define a plane, you can calculate the surface normal.

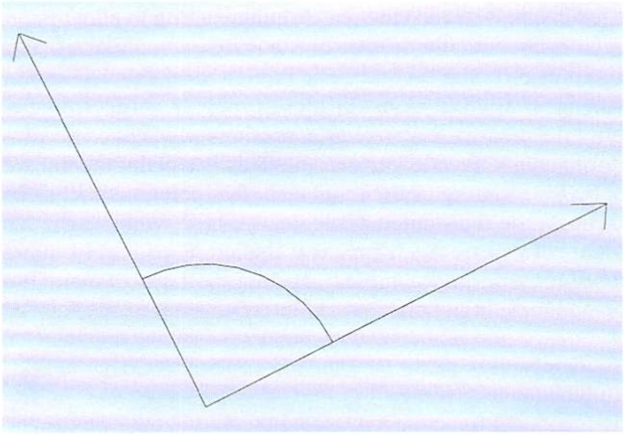


Figure 9: Find the angle between the vectors to determine the polygon's shade.

The next function in the display process calculates a surface normal for each of the object's polygons. Remember that surface normals are vectors that lie perpendicular to a plane in 3-D space. Polygons are essentially planes on their own, consequently they have surface normals associated with them. These surface normals are very important because they can be used to describe the direction a polygon is facing—necessary information for polygon shading and backface removal. The calcnormals( ) function loops through each of the polygons and calculates their surface normals using the vector cross-product operation. This operation finds a vector that is perpendicular to a plane defined by two vectors (it takes at least two vectors to define a plane). The first operation we perform inside the loop is calculating the two vectors we will use for each polygon. To determine the first vector we subtract the polygon's second vertex from its first. For the second vector, we subtract the last vertex from the first, as well. This way, both vectors originate from the first polygon vertex (see Figure 8).

After calculating normals for all of the polygons, we need to calculate their centers. We use a rather simple function that loops through all of the polygons and finds the average of each X, Y, and Z coordinate for each polygon. It then stores the average point as the polygon's center. The polygon centers are used for both shading and sorting, both of which are next up in the display process.

SHADY MANEUVERS

To find shading values for each polygon, the program calls calcshading( ). This function works by taking advantage of the vector dot-product operation, which finds the cosine of the angle between two vectors (see Figure 9). As the function loops through each of the polygons, it first finds the vector that extends from the light source to the polygon's center. It then uses this vector and the polygon's surface-normal vector in the dot-product operation (nx, ny, nz is the polygon's normal):

```
dp=-lx * nx+-ly * ny+-lz * nz;
```

The resulting value is the cosine of the angle between the two vectors. This value will range from 1.0 to −1.0 (which corresponds to 0° to 180°). Because a negative value means that the angle is greater than 90° (which also means that the polygon is facing away from the light source), we change all negative values to zero. The next line of code converts this value between 0.0 and 1.0 into a value between 0 and 7:

```
polys[i].sha=dp * 7;
```

The resulting number is then used as the polygon's shade value. Remember, we set up eight gray colors in the palette for shading. The shading value, when added to 8, corresponds directly to those grays in the palette. A value of 0 is the darkest shade (black), while 7 is the lightest (white).

The last step before drawing is to sort the polygons. This ►

is necessary because we want to draw the polygons in order from those farthest away to those closest to the camera. If they were simply drawn in the order they appeared in the Polygon buffer, the object would be drawn wrong. The idea is that if polygons farthest away are drawn first, those closest would cover up those farther away and thus be the most visible. This is called the Painter's Algorithm, because the technique simulates the methods of a traditional painter, painting objects farthest away first. As the painter works towards the foregrounds, he can easily cover the background scenes to achieve the desired effect of depth.

For our purposes, we will use the QuickSort sorting algorithm, for which we need a function that compares the distances of two polygons. The compare( ) function calculates the distances of the centers of the polygons, then compares the distances and returns values as required by the QuickSort routine. Most C compilers supply a qsort( ) function; check your compiler's documentation if you have any problems.

At last, the object's polygons are ready to be displayed on the screen (after we set the background color to black). Note that there are two drawpolygons( ) functions, one for drawing a wire-frame display and the other for a solid display. As drawpolygons1( ) is the simpler of the two, let's look at it first.

Looping through all of the polygons, drawpolygons1( ) draws them using the Amiga Move( ) and Draw( ) functions. First, however, the routine checks to ensure all of a polygon's projections are within the bounds of the screen. You could write a function to clip polygons that fall outside of the screen's boundary, but because such a routine is beyond the scope of this article, we'll just refrain from drawing any polygons that lie outside of the screen either totally or in part. Notice how we're using the Projection array with the polygon's vertex array (*vtx) values. If you recall, we have been allocating and maintaining the same number of projections as there are vertices. Because there is an equivalent projection for each vertex, we can simply refer to the projections at drawing time instead of the vertices. If a polygon is sufficiently in bounds, we next set the color with the ROM Kernel's SetAPen( ) function, then draw the polygon by looping through each of its vertices/projections. The first point is specified again to close the polygon. The process repeats until all polygons have been drawn.

While similar, drawpolygons2( ) has a few important differences. It too checks to see if a polygon is in bounds, but it also checks if the polygon is a "back face." Because this routine draws the polygons "filled in," the object will appear solid. As an optimization, we can leave out any polygons that "face away" from the camera. We determine this simply by using, once again, the vector dot-product operation.

The backface( ) function checks the angle between the vector from the camera to the polygon's center and the polygon's surface normal vector. If the value is less than zero (angle greater than 90°), then the polygon faces away so we can skip it. If a polygon survives both tests, we set its color with the SetAPen( ). In addition, we also set the polygon's shade with SetBPen( ). Instead of drawing the polygons with a solid pattern, the program draws in a checkerboard pattern, using the APen color in half of the pixels and the BPen color in the other half. This results in an effective mixing of the two colors to produce a simulated change in intensity of the APen color.

After the colors are set, the polygon is drawn by calling the ROM Kernel functions AreaMove( ) and AreaDraw( ). Instead of specifying the first point again at the end of the polygon, however, we simply call the Amiga's AreaEnd( ) to finish the polygon.

That's it! The object is now visible on the screen!

All that's left is the tidying up: We simply call the ROM Kernel's Wait( ) to wait for a keypress. When one comes in, the program frees everything allocated, closes everything opened, and exits.

FEELING AMBITIOUS?

Although this program has its complex areas, we've only scratched the surface of 3-D computer graphics. On the lighter side, you could add a clipping routine to allow polygons to lie partially outside of the screen's boundaries. You could also fix the second basic limitation of this 3-D object viewer: If any polygons (or more precisely vertices) lie behind the camera, the projection routines will choke and produce some very odd projections. This is usually taken care of by the clipping routine that also clips polygons that poke through the film/screen plane (in this case we used the XY plane). If you are ready for a real challenge, you could create more realistic (and complex) shading methods to make the objects look even better. ■

Brian Wagner, one of the founding members of Cryogenic Software, is the developer of 3-D Professional. Contact him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or via PeopleLink (CRYO) or CompuServe (72137,573).*

QUARTET

THE DIGITAL MUSIC MACHINE

With Quartet your AMIGA becomes a 4-voice polyphonic synthesizer and a 4-track recorder.

WITH QUARTET YOU CAN:

$59.95

Compose music on the four scrolling staves by either input from the mouse or by recording a live performance played on the AMIGA keyboard or MIDI instrument, connected to your AMIGA via a MIDI interface to the Serial Data Port.

Choose your instrument from a choice of 100 musical instruments and sounds included with Quartet or import your own sounds from a sampler such as Microdeal's own A.M.A.S. cartridge.

MichTron

3201 DRUMMOND PLAZA
NEWARK, DE 19711
PHONE: 302 - 454 - 7946

Sapphire 68020 / 68881 Accelerator

Fits in the Amiga 1000, Amiga 500, and the Amiga 2000.
Fits snugly in the 68000 socket.
Easy installation - Included is a disk with installation instructions, pictures, and public domain benchmark software. Also included is a static safty strap and static safty instructions that apply to all computer hardware. Factory installed 12 MHz 32-bit 68020 CPU and a factory installed 12 MHz 68881 FPU. Speed increases of up to 2.4 times faster in standard interger processing, and up to 3.2 times faster in floating point. Small compact size makes the Sapphire the smallest accelerator yet! Only 3 1/8" x 4 1/4" x 1/2" total size. Not a psuedo accelerator, but a true 32-bit accelerator card using true 32-bit processors. A full one year factory warranty. Retail Price $399.95

MRBackup Professional

MRBackup Professional is the first full featured backup system for the Amiga utilizing the full potential of the Amiga! With over 60 built in ARexx™ commands, MRBackup Professional gives the user the ability to reach beyond standard backup options. This is the first full featured hard drive backup system that has floppy and SCSI streaming tape capabilities.

- Will backup to floppy or SCSI streaming tape - Fully tested with Commodore, Supra, Xetec, CLtd, Trumpcard, and GVP contollers.
- Full ARexx Integration - Over 60 usable commands.
- Utilizes the option to use standard AmigaDOS or Fastdisk Format.
- Has a user selectable file compression - selectable up to 16-bit.
- Uses full AmigaDOS intuition for ease of use.
- Floppy users can use up to 4 floppys.
- MRBackup is compatible with AmigaDOS versions 1.3 and 2.0
- Full SCSI Drive kit Available - call for price!!

Retail Price 54.95

Workbench Management System v2.0

The Workbench Management System (WMS) is a revolutionary idea in software for the Amiga! WMS is based on a button concept where a single click of the mouse button will launch your applications. Buttons can be assigned to any program on a hard disk, floppy, or network. WMS allows you to launch as many programs as your memory will allow for as fast as you can click the mouse. WMS also includes eight built in programs.

- Memo-ed: A text editor.
- Calendar: A calendar that you can add daily appointments to.
- Remind: Is built into the calendar and can be set up to remind you upon boot up every day.
- Telemate: A phone address book with dail capibiltiy.
- Squeezebox: Archiving with the compression programs made simple. And More!

Retail Price $49.95

Teachers Toolkit

Teachers Tool Kit is a complete lesson planner and grade book written by a teacher for the teacher. Teachers Tool Kit allows the instructor to plan the lessons for the class and track the advancement of each student seperatly, or each different class as a whole. Some of the built in features of Teachers Tool Kit Are:

- The ability to store and print graphs, reports, grade book and student information.
- The ability to assign each student a unique ID number.
- The ability to create general subject areas that would have seperate lessons, tests, quizs, homework, and project that need to be completed by indivdual calsses.

Retail Price $49.95

RxTools

RxTools is an object oriented user interface builder, which extends the capabilities of ARexx™. It connects the inate texual interface of ARexx™ with the user interface of the Amiga, more commonly known as intuition. Intution in its raw form is a difficult programming enviroment. RxTools converts intuition into a manageable system with minimal loss of flexibility. Furthermore, it makes it directly accessible in this form to the ARexx™ programmer. With the bult in text editor RxTools provides a complete development enviroment, which may be the best available for the Amiga, in terms of overall capability and ease of use. RxTools is a function host program, which when run, stays in the background and allows for the use of the future RxTools extension set to be utilized in ARexx™ scripts. RxTools provides intuition capibility including windows, gadgets, requestors, and more.

Retail Price 54.95

Brigade

A new revolution in gaming software for the Amiga. Most war games work on a turn by turn basis. Brigade takes you one step closer to reality by adding in real time action. Brigade offers excitment not found in other war game simulators. If you do not pay attention or you leave without pausing the game the computer will continue with its war plans and you could lose the game. You may take a break but the enemy never does.

- Real-Time game play.
- Built in scenario/campaign editor - you can create or modify any vehicle, weapon, aircraft, map, and more!
- Oversized map to allow for larger than one screen play.
- Full digitized sound.
- Animation of weapons firing.
- Full control of units, their orders, and missions.
- Special Limited Time Offer - Desert Shield Data Disk Included!! •

Retail Price $44.95

TTR Development, Inc.

1120 Gammon Ln.
Madison, WI 53719
(608) 277-8071
United States, European, Australian Sales and Distribution

Pre'spect Technics, Inc.

P.O. Box 670, Station 'H'
Montreal, Quebec H3G 2M6
(514) 954-1483
Canadian & South American Sales and Distribution

AmigaDOS C Development System

Aztec C68k/Am-d Developer System

The battle continues. . .

By David T. McClellan

More commonly referred to by their manufacturers' names (SAS and Manx), these two rival compilers have been trying to outdo each other for years. SAS/C (formerly Lattice C) was the first C compiler available on the Amiga and was shipped in native, PC, and Sun cross-compiler versions with the original Amiga 1000 developer kits. The official, Commodore-sanctioned compiler, it had the field to itself for about a year, then Manx C came along, generating smaller executables and providing some other advantages. The challenge of opposition has prompted the two to continually improve code quality, execution speed, and tool quality.

At this writing, the current versions are 5.10 for SAS/C and 5.0a for Manx (although generation six should arrive shortly). Both compilers are multiple-pass and come with a shell to automatically invoke each pass. Both have optimizers that do a reasonable job of making code faster after you debug it (see the benchmarks, below). As you can see in Table 1, they are packed with features. For porting purposes, each compiler comes in an MS-DOS version, as well. While Manx also offers a Macintosh compiler, SAS offers one for OS/2.

ALL INCLUDED

SAS and Manx both provide versions of preprocessed headers (include files), but each compiler handles them differently. Offering 1.3 and 2.0 includes, SAS/C compresses the header files in a data-model-independent fashion, and the distribution disks come with precompacted headers for the SAS and Amiga libraries. SAS also provides lcompact, a tool that compacts new header files that you write or download (such as the IFF headers).

Manx C takes a different route: The compiler can precompile header files and write the symbol table information out to disk to load directly during later compiles. This symbol table information will vary for different compiler switches (large or small data models, int sizes, and other factors), so Manx does not provide a precompiled set on their distribution disks. Pick the compiler switches you will use, precompile the headers your program will be accessing with those switches on, and use the result. The Manx approach requires a little more effort on your part.

Table 1: Compiler Features

Feature	Manx	SAS/C	Notes
ANSI Standard	X	X	See the note on SAS/C keywords, below.
68020/030	X	X	
68881/2 Math	X	X	
IEEE Math	X	X	
Motorola FFP Math	X	X	
Optimize for Space		X	See the note on Manx optimizations, below.
Optimize for Time		X	
Precompiled Includes	X	X	
Small and Large Memory	X	X	
Prototype Generator	X	X	Generate ANSI function prototypes from your source files.
Compiler Pragmas	X	X	Compiler options embedded in code.
#asm Support	X		
Cross-reference Gen'd		X	

SAS/C provides extra keywords to aid Amiga programmers. For example, the keyword chip marks a data structure as belonging in chip RAM when loaded. SAS/C also offers some 80x86 memory-model keywords that have minor roles on the Amiga (near, far, huge) and a few having to do with the function declaration all (_regargs, _stdargs, _asm, _interrupt). You'll need these, however, in specialized applications only.

Both compilers provide extended #define names that you can use in conditionally compiled code, using ANSI-style shapes for those names. Manx, for example, defines _INT32 if ints are 32 bits wide (versus 16) and if you choose certain memory models it defines _LARGE_CODE and _LARGE_DATA.

Unfortunately, Manx C does not have a single compiler switch that tells the optimizer to select time versus space optimizations. By picking and choosing among the -pXX and -sXX options, however, you can instruct it to use certain optimizations and skip others in effect choosing time or space optimization by hand.

EDITORS, DEBUGGERS, AND MORE

SAS/C and the Manx C developer package come with several "extra" programs in addition to the standard macro assembler, editor with some degree of compiler integration, object librarian, linker, source-level debugger, and good make program. The hard-drive installation programs let you customize your set up.

Manx's editor, Z, is similar to the vi editor on Unix, including a few of its ex commands, macros, and a ctags

Table 2: Debugger Features

Feature	Manx	SAS/C	Notes
Breakpoints	X	X	
Conditional	X	X	
Repetition Count			
Variable Watch	X	X	SAS watches variable or memory range, Manx just memory range.
Single Step	X	X	
Over Routine Call	X	X	
Into Routine Call	X	X	
In Assembly Listing	X	X	
Variable Exam	X	X	
Structs/Unions	X	X	
Pointer Dereference	X	X	
C-style	X	X	
Assign Value/Expression	X	X	
Stack Traceback	X	X	
Disassemble Code	X	X	
Display Registers	X	X	
Display Memory	X	X	
Command Macros	X	X	SAS does this via ARexx interface
Online help	X	X	

Table 3: C Library Functions

Group	Manx	SAS/C	Examples
Math	31	40	float & int & error reporting
StdIO	40	40	fopen, printf, access
Low-Level IO	10	12	open, creat
Memory Mgmt	11	25	malloc, sbrk
Time/Date	8	12	asctime, getdate (includes two SAS/C "string" functions
Conversions	11	23	atol, gcvt, sscanf
OS, Register	16	36	chdir, geta4, signal (SAS includes seven Amiga file-system-specific I/O functions)
String	30	69	strlen, isupper (SAS includes eight file-pathname parsing functions)
Varargs	3		
Miscellaneous	12	16	assert, exit, qsort

program to generate indices to routine names for quicker editing. Z does not run the C compiler, but it can be invoked by the compiler's QuikFix option to fix syntax errors and restart the compiler. While the manual mentions that you can set up other ARexx-callable editors to work with the compiler in a similar fashion, it does not describe the interface very well. The Manx linker can create segmented executables (similar to overlays) to let you run executables larger than available memory. The object-module librarian and the assembler do their jobs well enough.

Manx also provides diff (a source-file comparer), grep (multifile string search), set (sets a different kind of environment-variable model than the CLI provides; used by the compiler), ls (Unix-style directory lister), hd (a hex-file dumper), and numerous other minor utilities.

The SAS/C editor, lse, is a standard programmer's editor with mouse support (cursor move, text scroll, pull-down menus). It can invoke the compiler directly and can be called via ARexx. Because you can use all the normal editor commands and a few special additions from ARexx, you can create some powerful macros. The SAS linker, blink, has a very good overlay-generation facility and can take its directives from a response file. The object librarian and assembler are the standard of their ilk.

SAS/C also provides versions of diff and grep, cxref (a multifile cross-referencer), lprof (an execution profiler), omd (an object-module disassembler), and other utilities.

Both packages come with example source programs, none of them complicated, but some useful. The Manx developer system includes the sources to the C library routines; if you are having some porting or speed problems with a program you are writing these could be very useful.

Table 2 contrasts the features of the SAS CodePRobe (CPR) and Manx SDB symbolic debuggers. I found both adequate to my needs, although executables with embedded debugging information are large for both products. The SAS debugger has some support for debugging an application consisting of several tasks but you have to be careful switching among them. CPR has a few other peculiar commands, such as one to return from a function call before the function normally would, but this multitask debugging is an interesting direction. Manx's SDB has a feature that will execute an arbitrary C expression while debugging; this can set variables to expressions and do other interesting things. I doubt, however, you could enter a large, multiline loop this way.

Both debuggers provide all the features listed in Table 2, but in different ways. For example, you can use the SDB feature that allows you to execute an arbitrary C expression to assign variables or to call a function by hand. SAS's CPR provides separate commands to do the same two operations. Manx has a macro facility, SAS has an ARexx interface. You may have to fid- ►

dle around with a couple of commands in either debugger to do what you want.

The compiler and tools are only half the reason to buy a language package; the other half is the library functions provided. Besides the standard ROM Kernel and AmigaDOS Exec functions, SAS/C and Manx C have rich C libraries. Both have a full complement of ANSI/Unix-C standard I/O functions, math and string functions, and others. (See Table 3 for a complete list.) SAS also has some add-on libraries, such as a DBase III-file-compatible C library. Scan the public domain and you will find a number of graphics, IFF, and clipboard I/O routines (among others) for both compilers.

The manuals for both packages are large and fairly complete. Both have complete descriptions of the tools and their parameters, the debuggers, and their C libraries. Manx puts all their documentation in one large paper-bound book, whereas SAS provides two D-ring binders full of documentation, with index tabs and a master index. I prefer the SAS approach.

NUMBERS DON'T LIE

Fancy features are attractive, but cannot make up for a slow compiler. Curious how the latest incarnations of the two Cs would time out, I collected some benchmarks. The test suite I used consists of ten function sets originally compiled by John Hennessy and Peter Nye: Perm exercises routine call speed (heavy recursion), while Towers (towers of hanoi) works on routine calls and structure- and array-indexing. Queens (an eight-queens solver) tests argument passing, comparisons, recursion, and looping. IntMM (an integer-array multiplier) and MM (a floating-point version of IntMM) put integer math, floating-point math and array processing through their paces, respectively. Puzzle hits on all types of integer comparison and computation. The three sorts (QSort, Bubble, and Tree) test comparison and pointer usage. FFT (a fast Fourier transform) thoroughly exercises floating-point math operations. The times in Table 4 are measured in milliseconds, using Intuition's ClockTime( ) function and the Amiga's clock. Each is the average of three executions of the suite. The large data model was used for all versions and both compilers (32-bit ints, 32-bit pointers), because of the size of some of the data objects in the test programs.

As Table 4 shows, in some cases the SAS/C global optimizer greatly improved the times, in others it did no better than unoptimized code. If you're using the optimizer, time major sections of your code both before and after with the global optimizer to decide where it will help you the most and where it is a hindrance.

As you can see, in most of the tests SAS/C is somewhat faster; the exception is in the unoptimized integer-matrix multiply (perhaps the loop computations are to blame). The optimized IntMM jumps way ahead. SAS/C scores better overall in Puzzle, which performs a lot of integer math, array indexing (using array elements as other array's indices), and element comparisons, and in the IEEE floating-point tests. The exception is with the 68881 floating-point code, where Manx is a little faster (run 5).

Both Manx and SAS/C are good buys. SAS/C on the average generates somewhat faster code, has a few more standard routines, and has strong PC and OS/2 companion compilers. Plus, all the examples in the Amiga ROM Kernel and hardware reference series are written for SAS/C. Manx runs on more platforms and has a loyal and somewhat larger following among the public-domain and shareware developers. Both have good linkers, editors, symbolic debuggers, and other support tools.

Personally, I like SAS/C's editor and

Continued on p. 46

Table 4: Benchmark Times (in milliseconds)

Model/Compiler	Test Times									
	Perm	Towers	Queens	IntMM	MM	Puzzle	QSort	Bubble	Tree	FFT
1) Manx	4611	5217	2917	7517	23844	32934	2277	6956	4278	51778
1) SAS	4178	4783	1972	10195	22300	24417	1511	4173	3117	38050
2) Manx	3778	4511	1939	6450	22689	22728	1633	4917	3883	49434
2) SAS	4211	4244	1978	3906	15878	10167	1067	2211	3077	34484
3) Manx	661	945	472	1072	4344	4411	333	906	606	9366
3) SAS	572	839	267	1800	4450	2967	239	506	506	7756
4) Manx	500	839	233	911	4105	2911	233	633	533	8728
4) SAS	533	778	233	806	3433	1377	172	300	467	6994
5) Manx	505	833	267	500	700	2883	133	633	470	1339
5) SAS	567	667	233	300	800	1367	100	267	411	1406

Following are the five compile-and-run scenarios of the suite:
1: 7-MHz 68000 (Amiga 1000), no 68881, compiled with no optimizations
2: 7-MHz 68000, no 68881, compiled with optimizer (-mt -O for SAS, -so for Manx)
3: 25-MHz 68030 with a 68882 (Amiga 3000/30), no optimization
4: 25-MHz 68030 with a 68882, compiled with optimizer as in run 2
5: 25-MHz 68030 with a 68882, compiled with optimizer, 68020/030 code generation, and 68881/82 math libraries (-m2 -f8 -mt -O for SAS, -c2 -f8 -so for Manx, both linked with the appropriate math libs)

(My thanks to John and Mary Ellen for the use of their Amiga 3000.)

On Disk

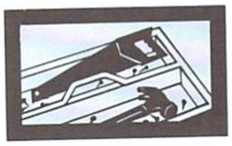
The AmigaWorld Tech Journal Disk

This nonbootable disk is divided into two main directories, *Articles* and *Applications.* Articles is organized into subdirectories containing source and executable for all routines and programs discussed in this issue's articles. Rather than condense article titles into cryptic icon names, we named the subdirectories after their associated authors. So, if you want the listing for "101 Methods of Bubble Sorting in BASIC," by Chuck Nicholas, just look for Nicholas not 101MOBSIB. The remainder of the disk, Applications, is composed of directories containing various programs we thought you'd find helpful. Keep your copies of Arc, Lharc, and Zoo handy; space constraints may have forced us to compress a few files.

All the supplied files are freely distributable—copy them, give them to friends, take them to the office, alter the source if it's provided. Do not, however, resell them. Do be polite and appreciative: Send the authors shareware contributions if they request it and you like their programs.

Before you rush to your Amiga and pop your disk in, make a copy and store the original in a safe place. Listings provided on-disk are a boon until the disk gets corrupted. Please take a minute now to save yourself hours of frustration later.

If your disk is defective, return it to AmigaWorld Tech Journal Disk, Special Products, 80 Elm St., Peterborough, NH 03458 for a replacement.



TNT

Technical News and Tools from the Amiga community.

Compiled by Linda Barrett Laflamme

Trend Setters

Dedicated to easy exchange of data between programs, a group of hardware and software developers founded **GRAFEXA**, "GRAphics EXtensions for the Amiga." GRAFEXA plans to provide a forum to discuss and set requirements for devices that handle more colors or higher resolutions than standard Amiga displays. To keep members in touch, the group is circulating a newsletter. For more information, contact Martin Lowe, Amiga Centre Scotland, 4 Hart Street Lane, Edinburgh EH1 3RN, Scotland, 031-557-4242 (voice), 031-557-3260 (fax). To get on the mailing list, contact Uwe Trebbien, Commodore Büromaschinen GmbH, Lyoner Straße 38, 6000 Frankfurt 71, 069-6638-0 (voice), 069-6638-159 (fax).

Help for C

The Amiga C Club (ACC) wants to make you a better C programmer. The non-profit organization provides advice on and solutions to specific coding problems, as well as general program design dilemmas. ACC also publishes the disk-based **Amiga C Manual**. The latest version, 2.0 consists of 15 chapters, more than 100 examples in both source and executable form, and several utilities (with their source).

The price of the manual (£20, $35, or SEK200) also includes registration in the ACC. Besides the ACM, registered club members receive unlimited free help, updates to the manual for the cost of the disk and postage, plus access to a video digitizer and a sound sampler (send ACC the raw materials, and they'll convert them to digital form). If you have already acquired the manual from the Fred Fish library, but would like to register, the fee is only £15, $25, or SEK150.

The ACC requests payments in cash, explaining cashing foreign checks in their native Sweden is extremely expensive. For details on the club or manual, write to Amiga C Club, Anders Bjerin, Tulevagen 22, 181 41 LIDINGO, SWEDEN.

Upgrade Update

INOVAtronics is now offering revamped versions of a pair of programmers tools. **CanDo version 1.5** ($149.95, $40 for upgrade only) adds database functions, multiple windows, floating-point math, Amiga OS 2.0 support, multidimentional array, record variable, and script-local variable support; a KeyInput object, and ARexx micro-servers to its litany of features. In addition, the authoring system sports a new script editor and editor tools, improved ARexx control, and easier user-definable error handling. Compatible with OS 2.0 and the latest Manx and SAS compilers, **InovaTools 1 version 2.0** ($99.95, $34.95 upgrade) provides custom interface routines in linkable C code, system library, and source formats. For further details on either package, contact INOVAtronics, 8499 Greenville Ave., Suite 209B, Dallas, TX 75231, 214/340-4991.

On-Line Journal

In March *The AmigaWorld Tech Journal* went on-line with the **amiga.world conference** in the Amiga Exchange on BIX (Byte Information Exchange). Whether you want to discuss technical issues, catch up on the news, suggest or comment on articles, flame, or ask for programming help, check out the tech.journal topic. For more general application discussions, try the amiga.world/magazine area. If you are interested in becoming a BIX subscriber, call 603/924-7681.

Psst. . .

Run across any hot products, PD software, or news? Send it along to TNT, The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458.* ■

On Disk

Improved Genlock Handling

Super Denise gives you more control over what you don't see.

By Dale Luck

THE FIRST COMPUTER with built-in genlocking capability, the Amiga has always housed some very clever circuitry. From the first, users were able to set up smoothly scrolling text for crediting video clips, simple graphics and text for annotating laser disc images, and overlay menus and buttons to allow interaction with the underlying video information. With the advent of the new Super Denise chip and Amiga OS 2.0, you can do even more.

The two key components of genlocking have been a means of controlling the vertical and horizontal timing of the Amiga and a way of deciding, on a pixel-by-pixel basis, whether the computer-generated picture or a video image from an outside source is displayed.

TRACKING THE BEAM

The vertical and horizontal signals in the Amiga RGB port are normally generated by the Amiga to control the display on a connected monitor. These vertical- and horizontal-sync outputs can actually be turned around, however, and used as inputs to the Amiga. When you turn on your Amiga, it senses signals to see if an external device is putting information on these lines. If it does not find any, the software tells the system's chips to generate all the signals necessary for a color monitor to display the Amiga's picture. If, however, the Amiga senses that another device is sending pulses on the vertical- and horizontal-sync output lines, it leaves the lines as inputs and uses the incoming signals to control the basic vertical-frame and horizontal-line timing of the Amiga. This way, when a video camera connected to the Amiga generates the first few lines of the picture, the Amiga generates the top few lines of the graphics display, putting the displays in sync or genlocking them together.

The Amiga builds the color display you see on the monitor by scanning consecutive memory locations (known as bitplanes) that are processed by the Denise custom chip. The pixels from these bitplanes flow through Denise and access a Color Look Up Table (CLUT) to generate a 12-bit color. The CLUT inside Denise is 32-words deep, allowing you to generate at most 32 completely independent colors in a standard lo-res screen. At the same time that these pixels are moving through Denise, another signal is being generated, the Z(Zero)-detect signal. This signal is sent through the RGB port and tells the connected device that the color now coming out was generated by a pixel value of zero. The zero value is normally associated with the background color. All other pixel numbers (1–31) cause the Z-detect circuit to generate a (non-zero) signal. A typical video mixing/genlock device attached to the Amiga uses the Z-detect signal to decide whether to display a pixel from the computer or one from the video camera. By carefully constructing the computer image in memory, the programmer, artist, or user can have the live video image show through these areas of zero, or transparency. The rest of the image, containing pixels of colors other than zero, will have colors generated by the Denise CLUT.

FLAKES FOR THE FUTURE

My first experiments with this genlocking capability were in December 1985 when I tried to simulate a snow flurry. Amiga artist Jack Haeger had already created a color-cycling animation of snowflakes drifting down outside a window. If I erased the image in the graphic window and pointed a video camera out a window in my house, I could have a computer-generated wall and window and live video of the snow-covered lawn outside. Then I could draw snowflakes and animate them with color-cycling to create a snow storm. I thought it would be a simple job to adapt Jack's animation.

Try as I might, however, I could not make the snowflakes disappear properly and drift down the outside live video image. I finally realized, to make the snowflakes vanish as they drifted to the ground, I needed to control which colors were transparent and easily cycle them. I also needed to have more than one transparent color at a time. Frustrated, but not beaten, I kept these capabilities in the back of my mind in case Commodore ever upgraded the custom chips.

A second problem I wanted to fix was the Amiga's treatment of the border color. The Amiga forces the border image, (the image outside the normal generated display) to pen number 0, meaning that the border is *always* transparent. As a result, there is no way to hide what could be distracting or unwanted video unless you can fully define the vertical and horizontal overscan image.

In 1987, we finally got the chance to try again. The Amiga chips needed to be upgraded to support higher resolutions, such as Productivity mode (640 × 480 non-interlaced). Both Agnus and Denise were in need of significant changes, and, while we were at it, I suggested we add some new features. We also changed the Amiga graphics libraries to support the new modes, added new function calls, and corrected mistakes in the 1.3 and earlier versions of the graphics library.

Good news for A1000 owners, some of the feature additions required enhancements only to Denise. Because Denise has remained basically the same chip throughout all Amigas (unlike Agnus), you can access several of the new features on the A1000 if you replace Denise and upgrade to 2.0.

The first addition is the ability to make any, all, or none of ►

the colors in the palette transparent. We did this by adding an additional colormap bit to each of the 32 color registers. This new bit is the most significant bit in each of the entries of the CLUT. The layout of the bits in the CLUT now looks like:

```
TXXXRRRRGGGGBBBB
```

where T is the new transparency bit, X is reserved, R is red, G is green, and B is the blue bit. The system uses the new T bit as the Z-detect signal instead of comparing the pixel number to zero. (This is just what I needed to do the color-cycle animation over a live video image.) As a side effect, because the control bit is in the colormap registers, you can use it to make parts of sprites transparent, as well.

The second addition is the ability to specify a particular bitplane for a transparency mask. A whole bitplane full of Z-detect bits are sent through the Z-detect line at the same time the normal pixel colors are generated.

The third new feature is the separate control of the border. You can now specify whether the border is transparent or not, plus specify a particular pen number to use for the border color.

The final modifications are not really features, but are very important considering one other non-genlock-specific capability. The addition of SuperHiRes mode means that the generation of the Z-signal needs to run at the 35 nanosecond rate instead of the old 70 nanosecond rate. SuperHiRes mode generates pixels that rival professional, multithousand-dollar character generators used in the broadcast industry. To use the new 35 nanosecond pixels and make them overlay properly on the live video, the Z-signal had to match it exactly. You do not need to write special code to take advantage of this feature.

CONTROLS AND TAGS

At this writing, I do not know of any programs that make easy use of these new genlock features. The only way to use them is by creating your own software. Study the test program, border.c (in the accompanying disk's Luck drawer), as an example. It toggles the border control for the first screen and employs two new 2.0 concepts, TagItems and VideoControl( ).

Because the code uses some new features, you must include the new 2.0 include files and link with the 2.0 libraries. You do so via the SAS/C 5.10a compile line below:

```
lc -iinclude:/compiler_headers_2.0 -L+lib:amiga2.0.lib border.c
```

Several of the pre-2.0 functions and many of the new 2.0 Amiga ROM functions accept a variable number of parameters, which you can set to program-specific values or the presumed defaults. Some parameters, however, may not make sense given the context of the others. You can make functions more general-purpose, however, by providing more information in the arguments. (This also helps control the function population explosion as new features are added to the hardware and the software.) The arguments to these generic functions are typically a context or object and a set of actions. In the case of the VideoControl( ) function, the object is the extended ColorMap/VideoControl structure and the set of actions is passed in as a pointer to an array of TagItems.

Each TagItem is usually broken into two parts: a command and an optional argument for that command. For example, the TagItem for controlling the genlock border transparency characteristics is:

```
{VTAG_BORDERNOTRANS_SET, 0}
```

to turn it on and

```
{VTAG_BORDERNOTRANS_CLR, 0}
```

to turn it off.

Because a list of TagItems can contain more than one, the list must always contain a method of telling the routine that there are no more TagItems to process. Therefore, every array of TagItems must always have an EndTagItem. Without proper termination, TagItem lists could easily cause software to behave erratically.

Now that you know basic operation of the TagItem, you can understand why the construct is perfect for the VideoControl( ) function. Under 1.3, one of the problems with the way the Amiga dynamically modified the display to generate real-time animations was synchronizing a complex set of changes so that they happened simultaneously, instead of some things being delayed while others were updated. This produced such effects as false color aliasing, jumping sprites, and displaced bitplanes. Because programs had to call many different graphics and intuition routines, as well as several structures to modify the display at just the right time, doing so atomically was very difficult—until now.

The new VideoControl( ) function helps to solve this display problem. VideoControl( ) does more than just update the genlock characteristics of the display. It can also change the colors and specify, in an atomic fashion, almost any change to the ViewPort that you wish to make. No more poking at the bits in the Modes fields of the ViewPort structure!

The border.c program is very straightforward. After ensuring it is running under the proper version of software and hardware, the program looks at the parameters typed by the user. If it understands them, it sets up the TagItem structure for VideoControl( ). It then calls the routine DoVideo( ), which actually calls VideoControl( ), and then causes the system software to remake the Workbench screen with the proper new Copper lists. When the program is finished, it leaves the features active for the Workbench screen.

The next step is to write a program that lets you not only enable and disable this basic genlock feature, but also control it on a pen-by-pen basis. Take a look at pentrans.c in the Luck drawer; it builds on several bits of code from the previous example. The flow of pentrans.c is very similar to border.c. The main change is the acceptance of an optional third parameter, the pen number.

These two demo programs illustrate the basic features only. To actually make snowflakes fall in a color-cycled animation entails a considerably more complex program. Because color cycling requires changes to several colors in the palette to achieve the proper effect, however, using VideoControl( ) and TagItems is an ideal method. With them, you will be able to change the transparency of any number of pens in the colormap with a single atomic function call. After studying Example.3 on disk, you should be ready to tackle your own genlock color-cycle animation programs. ■

Dale Luck was a member of the original Amiga team and was a design consultant for the Enhanced Chip Set. He is currently president of GfxBase, makers of the XWindows System for the Amiga. Write to him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or contact him on BIX (duck).*

UTILITIES UNLIMITED OF OREGON, INC.

P.O.Box 532 North Plains, OR 97133
ORDERS TAKEN 24 HOURS A DAY AT (503) 647-5611 FAX LINE (503) 648-8992

SUPER CARD AMI II

A MUST FOR ALL AMIGA OWNERS

Now is the time to own the most powerful backup system that ever will be made. We have searched for a program that this software, hardware package can not backup, and it is yet to be found. Over 10,000 units sold!!! Please join our search.

NEVER BEFORE 100% BACKUP......

- Easy to use, mouse driven software.
- Most software backed up in 60 seconds!
- Transparent when not in use.
- Fits any Amiga, even the 3000! (Please specify when ordering).
- Backup any 3.5" disk (IBM, ATARI, MAC, AMIGA).
- No soldering required!
- Cross-country BBS support system (Call for # nearest you).
- Backup your original - the day you buy it!
- Super Card AMI II works on NTSC (60 Hertz) and PAL (50 Hertz) systems.
- Tested world wide to be the 'one and only' 100% backup system.

NOW ONLY

$99.95

Don't wait, one original lost can cost more than this backup system! We have a full stock on hand and your system can be on its way to you fast!
Remember, specify the Amiga you have when ordering. A500/1000/2000/2500/3000 using one or more external drives, or A2000/2500/3000/ with two internal drives.

AMI-II SOFTWARE UP-DATE

Now Available 1.0 Software

- Copier files that allow 60 second backup for most programs.
- Easy to use instructions.
- Save those programs onto the copier files for future use.
- Join our automatic up-date list, and never miss another up-date.

$9.95

AMI SUPER TRACKER

Have you ever wanted to know where problem tracks are located? Now, with super tracker AMI you can tell! This beautiful digital track display simply plugs into the last drive in your Amiga system (all Amiga computers will work). The head location (track) side (top or botton head) and where write protect position are all displayed.

NOW

$79.95

M.A.S.T. UNIDRIVE

For those of you on a budget. Now is the time to order this great looking, reliable and quiet drive.
We at Utilities Unlimited can offer this drive for only.

$99.95

ENHANCED UNIDRIVE

The only Amiga external floppy drive in the world that includes Digital Track Display, Hardware Write Protect Switch and Hardware Virus Protection System.
All this for only . . .

$149.95

Add $4.00 Shipping and Handling - Add $3.50 C.O.D. in U.S.A. only. Add $3.00 to all foreign shipments. VISA and MasterCard are accepted.

ATTENTION TO OUR VALUED CANADIAN FRIENDS . . .

NOW ORDERING PRODUCTS IS EASY. AND FAST DELIVERY. WE HAVE SET UP A DISTRIBUTOR JUST FOR YOU. PLEASE PLACE YOUR ORDERS BY DIALING. (519) 272-1528 OR MAIL ORDER TO:
P.O. BOX 311, STRATFORD, ONTARIO, CANADA N5A 6T3.
FOR ANY TECHNICAL ASSISTANCE DIAL (503) 647-9022
THURSDAY & FRIDAY 10:00 A.M. TO 3:00 P.M. PACIFIC TIME!!!

SUPER-CARD UTILITY PACKAGE

Copier Construction Set - Create copier files for Super-Card AMI II vl.0 software.

Disk Anaylzer - Display format and structure information of tracks. This will help determine which mode you should use with Super-Card AMI II.

Drive Speed Checker - Checks drive speeds of ALL drives.

Drive Alignment Checker - Checks drive alignment of ALL drives.

MFM Editor - Read & Write MFM data. Works in conjunction with Copier Construction Set to help create copier files.

$39.95

KICK BOARD

When Workbench 2.0 is released, it is estimated that only 67% of the existing software will work with it. Nearly all commercial games will not run under the new Kickstart ROM. That leaves the consumer swapping their ROMs back and forth in order to run various software. This is a terrible inconvience to the consumer.

Introducing, KICK-BOARD..... A simple to install board that replaces your ROM inside your Amiga computer. Remove your old Kickstart ROM from it's socket and place it in our board. Now, plug the KICK-BOARD'S ribbon cable into the empty ROM socket. That's it! You can add two additional ROMs to the KICK-BOARD besides your original. Giving you the total of three possible ROMs to use in your Amiga.

By simply moving the switch provided to one of the three positions, you can select one of the available ROMs. No more compatibility problems! By using a ribbon cable assemble, we have insured that this product will work with all processor accelerators, which generally cover the ROM socket completely.

Introductory Price

$39.95

BOOT DRIVE SELECTOR

Tired of that annoying "Clicking" that your drive makes when there is no disk inserted? Have you ever wanted to boot from one of your external drives? Did you know that some commercial programs (generally European games) actually require your external drives to be disconnected from your Amiga!!

Introducing, BOOT DRIVE SELECTOR.... A simple to install board that fixes all of the above mentioned problems for good!! This unit installs between your CIA chip and your internal drive. Once installed, the "Clicking" (which will eventually wear out your drive) will be a thing of the past.

What happens if your internal drive malfunctions? You are stuck without your computer! Not if you have this unit installed! Simply select which external you want to boot from and you again have a usable system. The external drive you select and your internal (DFO) drive actually "Swap" locations, allowing the normal usage of all drives.

No more removing your external drives for those programs that require that there be only one drive online. Simply flip the switch, and presto! All external drives are disabled! This product will pay for itself without question!

Now with Anti-Virus

$49.95

The NTSC/RS-170A Video Standard

Understanding the broadcast video standard is the first step toward complying with it.

By Scott Hood

BROADCAST STANDARD IS the catch phrase of choice on almost every video product's brochure these days, but what exactly does it mean? What specifications must the device's output meet to conform to the National Television Standards Committee's NTSC/RS-170A video standard for Composite Video Baseband Signals (CVBS)? The answer lies in a knowledge of the standard and signal testing.

Used in North America, Japan, and several other parts of the world, the standard for color composite video is officially called the M/NTSC standard and is outlined by the EIA (Electronic Industries Association) RS-170A specification. The composite video signal has a 1 Vp-p (voltage peak-to-peak) maximum amplitude (assuming a 75 ohm load) and is limited to a bandwidth of 4.2 MHz by the FCC (Federal Communications Commission). The luminance (brightness) and the chrominance (color) of a pixel are encoded into the signal, as are various timing pulses and signal amplitudes. The total signal amplitude is divided into 140 units called IRE (Institute of Radio Engineers) units (1 IRE = 7.14 mV).

POLITICAL ELECTRONICS

Instead of total innovation, the introduction of color to the black-and-white RS-170 video world brought compromise. The FCC required all color broadcasts to be viewable on black-and-white receivers, as well. (Sounds like the proposals for HDTV, doesn't it?) Engineers partly accomplished this by clever sharing of the 4.2 MHz bandwidth between the luma and chroma signals. They also modified some of the signals' timings within the older RS-170 (monochrome video) specification enough to encode the color information. The band-sharing approach, clever as it is, is also responsible for many of NTSC video's artifact problems, such as dot-crawl and cross-color moire patterns. Figure 1 shows a representation of the frequency spectrum for composite NTSC video and the band-sharing of the color information (represented by the color components I and Q) and the luma (or monochrome/black-and-white) information.

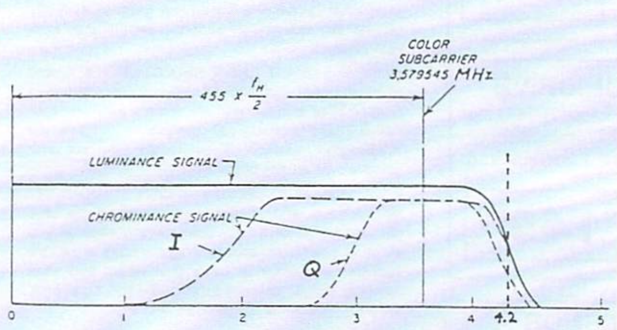


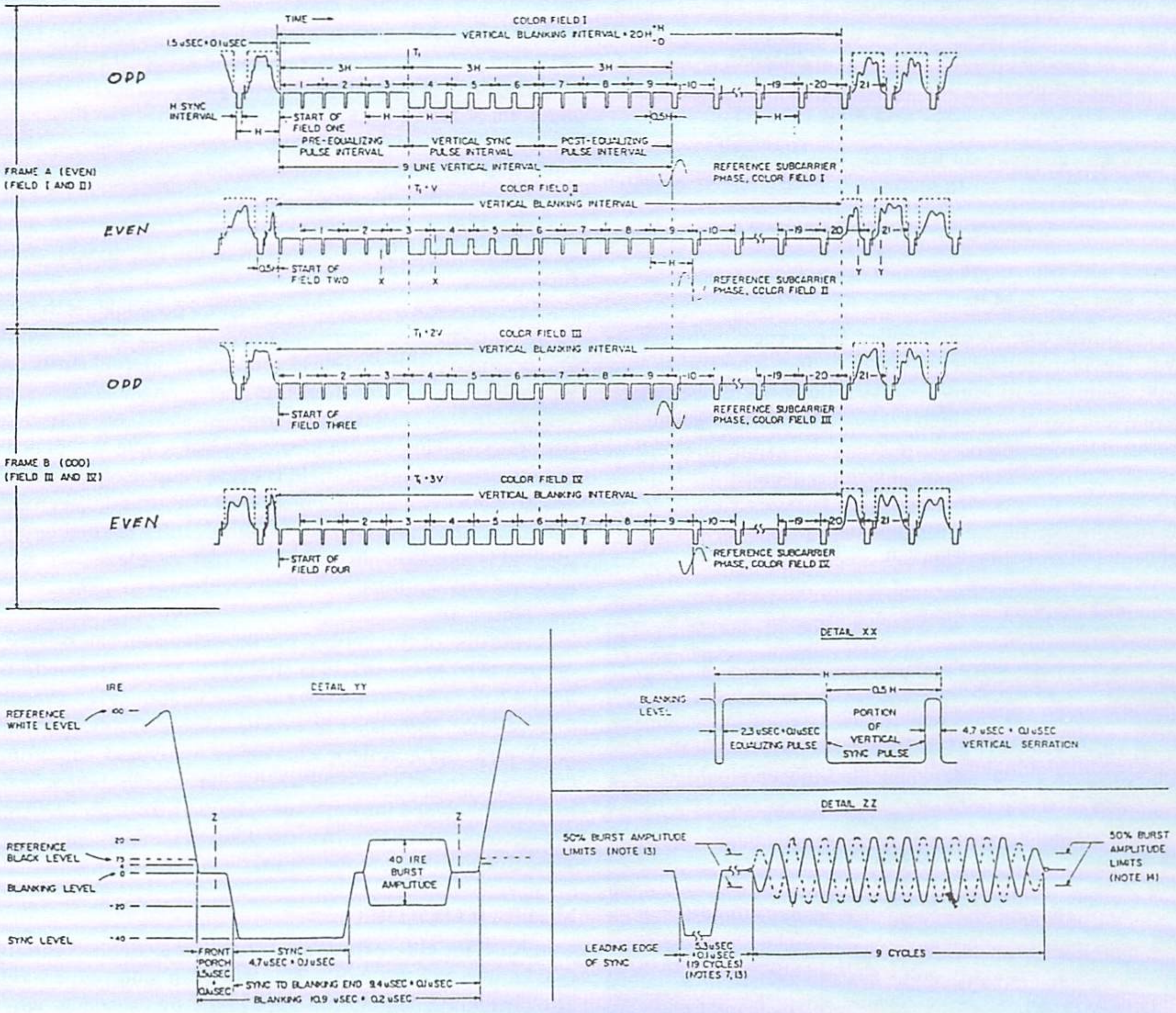
Figure 1: The frequency spectrum for composite NTSC video, showing band sharing of the color and luma information.

The video information of both color and monochrome broadcasts is presented in an interlaced format (called 2:1) to simplify broadcast and receiver systems. Again the compromise creates an unfortunate side effect: the all-too-familiar interlace flicker visible on horizontal lines and other high-contrast edges.

The signal's 4.2 MHz bandwidth also limits the resolution or the number of horizontal picture elements that can be encoded. The way you measure resolution in the world of broadcast television is different than the way you measure it in the world of computers. The 4.2 MHz-limited composite video signal contains nearly 330 "lines" of encoded information (using a rule of thumb of about 80 lines per megahertz). Compare this to the close to 3 MHz of bandwidth or 240 lines of information that common VHS VCRs can produce. Based on its allocation of 6 MHz per channel including the sound carrier, the FCC limits the bandwidth of the composite video signal (and therefore the number of resolvable lines in the display) to provide enough spectrum space for a large number of channels. Each separate composite video broadcast requires a channel for terrestrial broadcast. The composite video signal is modulated to the visual carrier frequency, (1.25 MHz) above the channel frequency, and the sound carrier is added 4.5 MHz above this.

SIGNAL CORE

The composite video signal itself is a time-based signal, where the display's basic elements are divided into fields (odd and even) and lines. Each field is made up of 262.5 lines (scanned left to right with 63.556 µS per line), where a portion of the field is used for blanking, vertical sync, and other control information. The two fields are interleaved such that the even field is followed by the odd field and then another even field and so on. Two fields, one even and one odd, mesh together to make up a frame of 525 lines in an interlaced for- ►


TIME
COLOR FIELD I
VERTICAL BLANKING INTERVAL • 20H
1.5 uSEC • 0.1 uSEC
ODD
H SYNC INTERVAL
START OF FIELD ONE
PRE-EQUALIZING PULSE INTERVAL
VERTICAL SYNC PULSE INTERVAL
POST-EQUALIZING PULSE INTERVAL
9 LINE VERTICAL INTERVAL
REFERENCE SUBCARRIER PHASE, COLOR FIELD I
FRAME A (EVEN) (FIELD I AND II)
COLOR FIELD II
VERTICAL BLANKING INTERVAL
EVEN
START OF FIELD TWO
REFERENCE SUBCARRIER PHASE, COLOR FIELD II
COLOR FIELD III
ODD
START OF FIELD THREE
REFERENCE SUBCARRIER PHASE, COLOR FIELD III
FRAME B (ODD) (FIELD III AND IV)
COLOR FIELD IV
EVEN
START OF FIELD FOUR
REFERENCE SUBCARRIER PHASE, COLOR FIELD IV
DETAIL XX
BLANKING LEVEL
2.3 uSEC • 0.1 uSEC EQUALIZING PULSE
PORTION OF VERTICAL SYNC PULSE
4.7 uSEC • 0.1 uSEC VERTICAL SERRATION
DETAIL YY
IRE
REFERENCE WHITE LEVEL
REFERENCE BLACK LEVEL
BLANKING LEVEL
SYNC LEVEL
40 IRE BURST AMPLITUDE
FRONT PORCH 1.5 uSEC
SYNC 4.7 uSEC • 0.1 uSEC
SYNC TO BLANKING END 9.4 uSEC • 0.1 uSEC
BLANKING 10.9 uSEC • 0.2 uSEC
DETAIL ZZ
50% BURST AMPLITUDE LIMITS (NOTE 13)
50% BURST AMPLITUDE LIMITS (NOTE 14)
LEADING EDGE OF SYNC
5.3 uSEC • 0.1 uSEC (19 CYCLES) (NOTES 7, 13)
9 CYCLES

Notes

Drawn: JWPG May 16, 1977

1. Specifications apply to studio facilities, common carrier, studio to transmitter and transmitter characteristics are not included.

2. All tolerances and limits shown in this drawing permissible only for long time variations.

3. The burst frequency shall be 3.579545 MHz ±10 Hz.

4. The horizontal scanning frequency shall be 2/455 times the burst frequency [one scan period (H) = 63.556 μSEC].

5. The vertical scanning frequency shall be 2/525 times the horizontal scanning frequency [one scan period (V) = 16.683 μSEC].

6. Start of color fields I and III is defined by a whole line between the first equalizing pulse and the preceding H sync pulse. Start of color fields II and IV is defined by a half line between the first equalizing pulse and the preceding H pulse. Color field I: That field with positive going zero-crossings of reference subcarrier nominally coincident with the 50% amplitude point of the leading edges of even numbered horizontal sync pulses.

7. The zero-crossings of reference subcarrier shall be nominally coincident with the 50% point of the leading edges of all horizontal sync pulses. For those cases where the relationship between sync and subcarrier is critical for program integration, the tolerance on this coincidence is ±40° of reference subcarrier. Reference subcarrier is a continuous signal with the same instantaneous phase as burst.

8. All rise times and fall times unless otherwise specified are to be 0.14 μSEC ±0.02 μSEC measured from 10% to 90% amplitude points. All pulse widths are measured at 50% amplitude points, unless otherwise specified.

9. Tolerance on sync level, reference black level (set-up) and peak to peak burst amplitude shall be ±2 IRE units.

10. The interval beginning with line 17 and extending through line 20 of each field may be used for test, cue and control signals.

11. Extraneous synchronous signals during blanking intervals, including residual subcarrier, shall not exceed 1 IRE unit. Extraneous non-synchronous signals during blanking intervals shall not exceed 0.5 IRE unit. All special purpose signals (VITS, VIR, etc.) when added to the vertical blanking interval are excepted. Overshoot on all pulses during sync and blanking, vertical and horizontal, shall not exceed 2 IRE units.

12. Burst envelope rise time is 0.3 μSEC ±0.2 μSEC measured between the 10% and 90% amplitude points.

13. The start of burst is defined by the zero-crossing (positive or negative slope) that precedes the first half cycle of subcarrier that is 50% or greater of the burst amplitude. Its position is nominally 19 cycles of subcarrier from the 50% amplitude point of leading edge of sync. (see detail ZZ)

14. The end of burst is defined by the zero-crossing (positive or negative slope) that follows the last half cycle of subcarrier that is 50% or greater of the burst amplitude.

15. Monochrome signals shall be in accordance with this drawing except that burst is omitted, and fields III and IV are identical to fields I and II respectively.

16. Horizontal blanking width is fundamentally based on the blanking of the picture signal. When the equipment that is used to insert set-up into a picture signal directly controls the blanking of the picture, then a measurement at 4 IRE units will also be a measure of the picture blanking. Frequently, the equipment that is used to insert set-up (such as a stabilizing amplifier) will not directly control the picture blanking, so that the blanking may be wider than that measured at 4 IRE units. In such cases, a measurement must be made at some picture level to establish the actual picture blanking, which must satisfy the dimension in detail YY. It is recognized that occasionally the picture content at the edge(s) of the picture may be black. Under such circumstances, the picture must be viewed on a properly adjusted picture monitor to verify that black edge(s) are consistent with the scene content and not the result of improper equipment adjustment.

Figure 2: RS-170A color television studio picture line amplifier output (SP 1281). Diagram reprinted with permission of the Electronic Industries Association.

mat. This means that the even-field lines are scanned on the display and interleaved with the odd-field lines. The beginning of one field is separated from the next by 16.683 mS (about $1/_{60}$ second). Thus a frame, which is made up of two fields, is 33.366 mS (about $1/_{30}$ second) in duration. These timings are determined by the following relationships to the color burst frequency of 3.579545 MHz (±10 Hz):

One scan line = 2 ÷ 455 × color burst frequency = 63.556 μS
One field = 2 ÷ 525 × (1 ÷ 63.556μS) = 16.683 mS

These values expressed in terms of frequency are:

Horizontal scan rate = 15.734 KHz
Vertical scan rate = 59.94 Hz

In each field one region is called the vertical blanking interval. This blanking interval is 20 or 21 lines long and, as you would assume by its name, contains the vertical reset pulse interval and no active video (at blanking level). At the beginning of the vertical blanking interval is a nine-line vertical interval, during which no color burst is present. Within this vertical interval all fields (even or odd) have a three-line region (the pre-equalizing pulse interval) before the actual vertical sync-pulse region. This region consists of pulses, 2.3μS (±0.1μS) in width, every half-line (or every 31.778μS).

Following this three-line region is the vertical sync pulse interval, which is three lines long total. This region consists of active-low pulses with 4.7μS (±0.1μS) active-high vertical serration pulses trailing each half-line group of pulses. After this region is the post-equalizing pulse interval, which is again three lines long for all fields. The pulses in this region are the same format as in the pre-equalizing pulse interval. Note that most all of the time measurements given are determined at the 50% amplitude points of the signal being measured. Also, all rise and fall times of these signals are to be 0.145μS (±0.02μS) measured from the 10% to 90% amplitude points.

The difference between the even and odd fields is the start of the vertical sync interval. Counting from the last whole-line boundary, if the vertical sync pulse interval starts on a whole-line boundary, then the current field is odd. Conversely, if the vertical sync interval starts on a half-line boundary (again counting from the last whole line of active video), then the current field is even. All pulses, unless otherwise noted, are active-low at −40 IRE units as seen on a waveform monitor. The non-active portion of the blanking interval is at 0 IRE, also called the blanking level. For reference, the maximum signal (active video) is at 100 IRE units (see Figure 3).

Part of the of the vertical blanking interval can be (but is not required to be) used for control, test, or cue signals. On lines 17 through 21, the signal may include VITS (Vertical Interval Test Signals), VIR (Vertical Interval Reference signals), and alphanumeric characters for closed captioning, and so on. The FCC also allows the vertical blanking interval to contain teletext information in lines 15 through 18. In studio environments, line 16 is used for VITC (Vertical Interval Time Code) to ensure field/frame accurate editing.

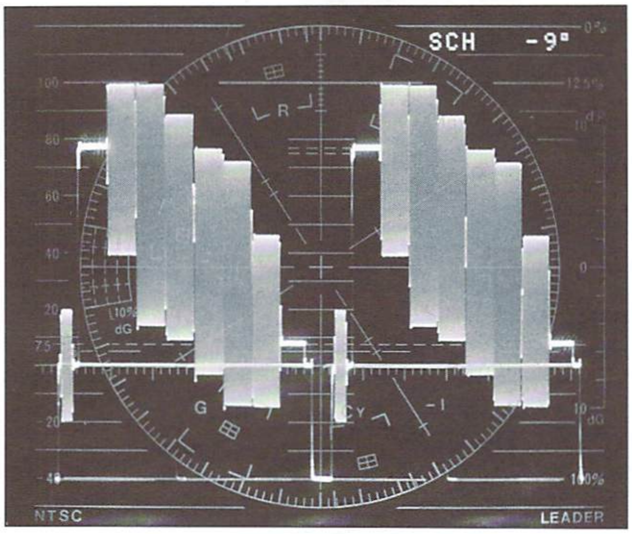


Figure 3: Full-field color bars as seen on a waveform monitor.

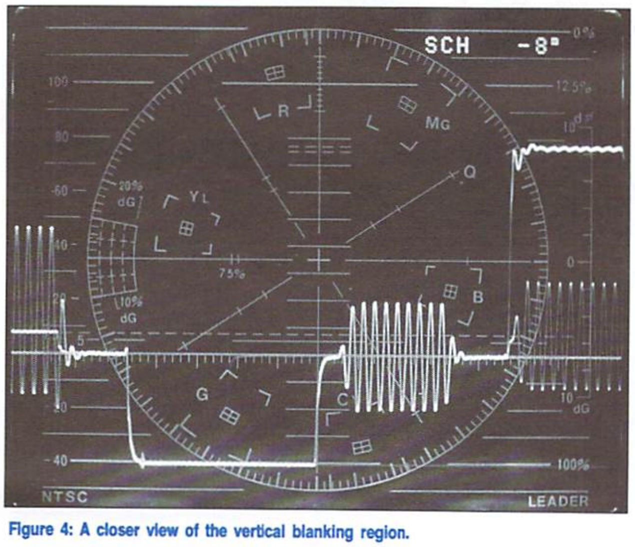


Figure 4: A closer view of the vertical blanking region.

MORE FIELD WORK

In RS-170A video, there are actually four fields (I through IV), known as color fields. Two are even, and two are odd. The differences among the color fields are caused by where the phase of the color subcarrier is in relation to the horizontal sync. Because of the 3.579545 MHz (±10 Hz) frequency of the color subcarrier (color burst), it needs four color fields to repeat. The RS-170A specification defines color field I as the field for which the rising extrapolation of the reference subcarrier burst intersects the 50% point of the horizontal sync's leading edge at the zero crossing of the subcarrier for even numbered lines. For accurate color framing, the allowed tolerance on this intersection is ±40 degrees. If the subcarrier-to-horizontal sync phasing (SC-H) is greater than ±40, the color framing is ambiguous. Accurate color framing is vital in an editing environment, because it helps prevent such video artifacts as color smearing and horizontal shifting effects.

LINE ITEMS

Depending on where they occur during the field, the lines that comprise a field may contain equalizing pulses, vertical serration pulses, vertical sync pulses, horizontal sync pulses, blanking information, or active picture information. The lines that are active (scanned lines that you can see) contain a horizontal sync pulse, horizontal blanking level, picture black level, color burst subcarrier, and the encoded luminance and chrominance information that defines a line of picture information (see Figure 3).

All active horizontal lines begin with an interval called the picture blanking interval, which is 10.9 µS (±0.2µS) wide (refer to Figure 4) and comprised of several regions. Starting from the left, the first is the front porch region, which is 1.5 µS (±0.1µS) wide and at 0 IRE units (the blanking level). Next comes the 4.7 µS (±0.1µS)-wide horizontal sync pulse itself at −40 IRE units. To its right is the breezeway region, which is typically 0.6 µS in width and at 0 IRE units. Measured from the leading (left) edge of horizontal sync, the color-burst sine wave starts next at 5.3 µS (±0.1µS). The color burst is a sine wave that is nine complete cycles of the 3.579545 MHz color subcarrier frequency. The color-burst sine wave's amplitude is limited to 40 IRE units peak-to-peak (+20 to −20 IRE as seen on the waveform monitor, Figure 4). The burst envelope is limited to a rise time of 0.3 µS (+0.2µS, −0.1µS). The final region in the picture blanking interval is the back porch, which is typically 1.58 µS in width and at 0 IRE units.

The RS-170A specification allows a tolerance on the sync level and color-burst amplitude level of ±2 IRE units. Extraneous synchronous signals are limited to 1 IRE unit, while extraneous nonsynchronous signals are limited to 0.5 IRE unit (test signals in the vertical blanking interval are excepted). The specification also limits overshoot on all pulses to 2 IRE units.

The rest of the 63.556 µS horizontal line interval is made up of active picture information and can range from −16 to +100 IRE units in amplitude. (Note that the color black, as transmitted during the active picture portion of the line is not at the blanking level of 0 IRE, but is specified at the "picture black" level of 7.5 IRE, which is sometimes called the black pedestal or setup.)

If all of this information seems a bit overwhelming, you are not alone. I suggest you consult the EIA RS-170A specification waveforms and notes in Figure 2 and the excellent *Television Engineering Handbook* (edited by K. Blair Benson, McGraw-Hill, New York, 1986), which I consider the bible of video engineering. You will be hard pressed to find a more comprehensive collection of facts and figures pertaining to the full spectrum of todays' television and video arenas. For an official copy of the standard's specifications (document IETNTS1), contact the Electronics Industries Association, Standards Sales Office, 2001 Pennsylvania Ave. NW, Washington, DC 20006, 202/457-4966.

SCOPE IT OUT

Specifications and diagrams only tell half the story. The remainder comes from testing and measuring a video device's output with a NTSC signal generator, a waveform monitor, a vectorscope, and an oscilloscope. The NTSC signal generator creates test patterns that you can channel through the video device in question. With the waveform monitor, vectorscope, and oscilloscope, you can then observe the degree to which the video device degrades the signal during processing and output. The signal generator is therefore a very accurate source for "perfect" or complete RS-170A compliant video. A waveform monitor allows you to view the composite waveform and perform several types of measurements on the video device's output, such as checking signal levels and amplitudes (see Figure 4). A vectorscope lets you observe and measure the color amplitude and phase of each color in the composite signal. Finally, you can use an oscilloscope to measure many of the timing aspects and voltage levels of the composite video signal (see Figure 5).

But what do the results mean? Let's look at the output of a properly adjusted NTSC RS-170A signal generator on each of the four basic pieces of test equipment.

Figure 3 shows a waveform monitor viewing two lines of video while the signal generator outputs the EIA RS-189A full-field color bars. Notice that the active-low horizontal sync pulse (shown splitting the center vertical graticule) for the second horizontal line (left to right) is at −40 IRE. To the right of the sync pulse you can see that the color burst (which appears as a dense band) is from +20 IRE to −20 IRE. The next major feature is that the color bars appear uniform and at the correct levels of chroma amplitude. You can even see the 7.5 IRE black setup level near the end of the line.

Figure 4 shows a different magnitude setting for viewing the same horizontal line as in Figure 3. At the very left of the figure, you can see the 7.5 IRE black level setup and following it the 0 IRE level of the front porch. Next comes the horizontal sync pulse. Notice how clean the horizontal sync is with well-defined transitions and the proper level. To the right of the horizontal sync are the nine cycles of color burst (count them) with the proper +20 IRE and −20 IRE levels in detail. ►

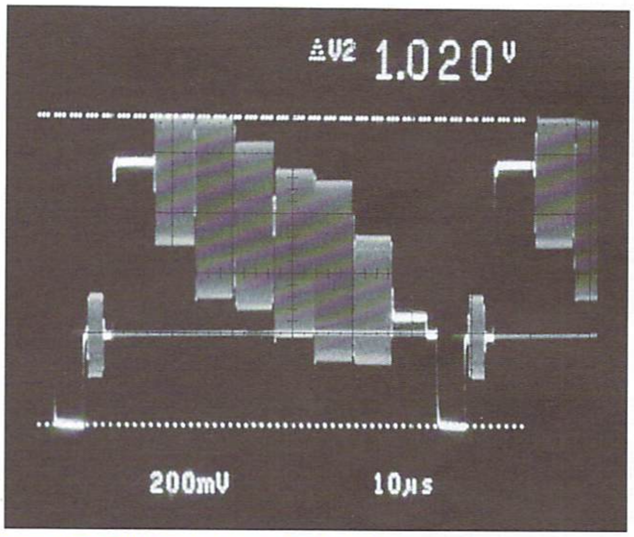


Figure 5: The peak-to-peak voltage output of the signal generator seen on an oscilloscope.

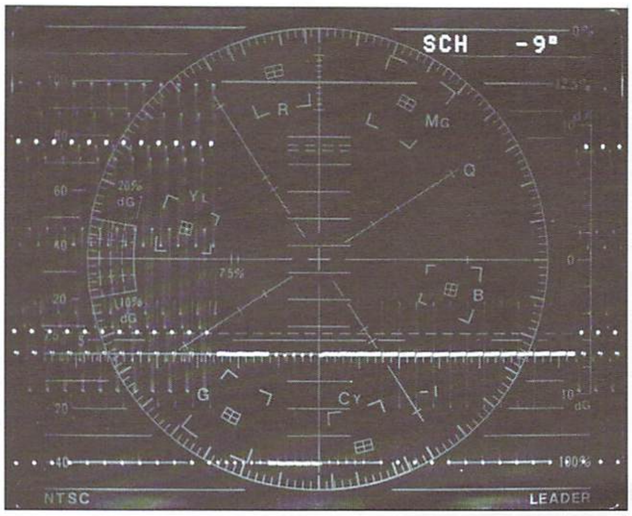


Figure 6: The vertical blanking region of a field.

In Figure 6, the waveform monitor displays the vertical blanking region of a field. Here you can make out the absence of the color burst on the first nine lines of video and clearly see the vertical sync interval.

Figures 7 and 8 illustrate a vectorscope's output (I use a combination waveform monitor/vectorscope). Notice that, in Figure 7, each color of the color-bar test pattern is represented (gray, yellow, cyan, green, magenta, red, blue, and black). The interconnected bright dots indicate the end of each color vector and thus both the phase angle and the color saturation, or amplitude. On the vectorscope, the correct phase and amplitude are represented by the small target box for the EIA colorbar pattern. If the signal has the correct encoding, the dots will be right on target. Note that the right corner of Figure 7 displays the SC-H sync phase. My waveform monitor/vectorscope provides an automatic measurement of this critical tolerance, which is within the ±40° limit of the RS-170A specification. Figure 8 is a vectorscope display of the EIA split-field color-bar test pattern showing the additional −I and +Q color component vectors.

The oscilloscope results are in Figure 5. I am measuring the peak-to-peak voltage output of the signal generator with a

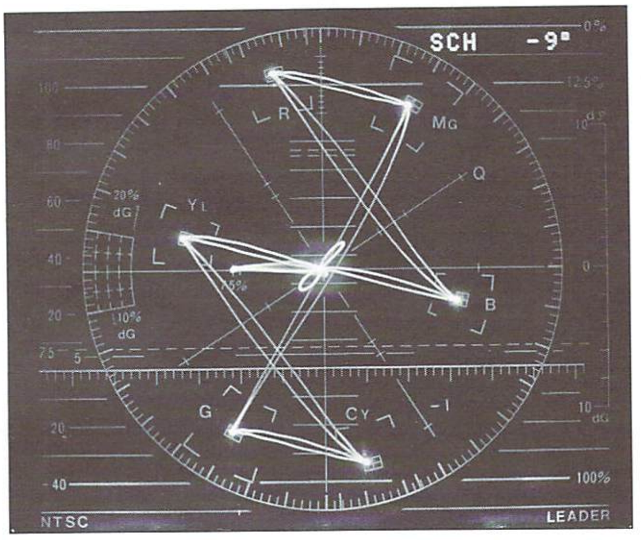


Figure 7: The EIA color-bar test pattern viewed on a vectorscope.

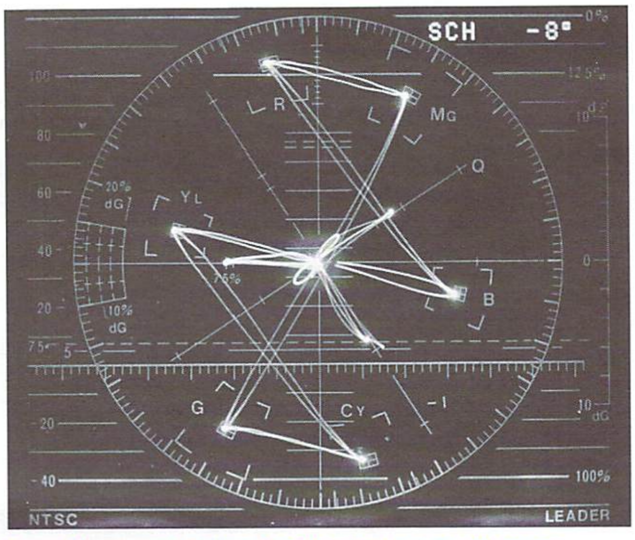


Figure 8: A vectorscope display of the EIA split-field color-bar test pattern.

75 ohm external load to determine if the signal is at the correct 1 Vp-p level. I can also use the oscilloscope to measure all of the critical timing tolerances for sync and burst rise/fall times, pulse widths, and so on. Using these pieces of video test equipment and others, you can begin to verify if a particular video device meets the RS-170A specification.

STRAYS

If you strictly apply the RS-170A and all its fine details (timing tolerances, IRE tolerances, rise/fall times, and so on), many (but not all) video cards that purport to output RS-170A composite video deviate from the letter of the specification. The devices may be in violation of such fine points of the RS-170A specification (rise/fall times on burst or sync, or the subcarrier-to-horizontal sync tolerance) that the aberrations will most likely not seriously affect the quality of the composite video output. The violations may, however, have other consequences depending on how and where (a studio environment) the devices are being used.

Incorrect video levels are a major problem that plague most of the low-cost genlock/encoder cards. These levels have to do with both the chrominance and luminance amplitude components of the output composite video signal. Having the incorrect levels can seriously affect the quality of the signal's image on your display, erode the quality of multi-generation tape recordings, and simply ruin the signal's usefulness in a studio environment. Sometimes you can, with the appropriate equipment, adjust the device to meet the RS-170A specification, but this is really not desirable.

Be on the watch for 3.58 MHz ringing, or noise, on top of the video signal itself, as well. This can most clearly be seen on a waveform monitor adjusted to look at the horizontal sync interval. The inaccuracy manifests itself to the user as excessive chroma-crawl (it looks like a moving zipper on most vertical edges) on the display and may cause other video artifacts.

A third problem with the low-cost genlock/encoder cards is the encoder circuit's accuracy at encoding both the proper phase and amplitude of each of the saturated colors. An encoder circuit (the circuit that creates the composite video for output) in this case will appear on the vectorscope to have a better response for one range of colors than another. For example, the colors from cyan to blue may be over-saturated and with greater phase shift when compared against colors from red to yellow. This will create distortions in the actual representations of the colors and degrade the image.

While the current market indicates that the likelihood of a device's abuse of the RS-170A specification is inversely proportional to the unit's price, this need not be the case. Designers, test and tweak your devices before they get into the users' hands. The increased sales to standard-savvy consumers will be worth the extra development time. ■

Scott Hood is a hardware design engineer with Commodore Technology Group, West Chester, PA. He designed the A2320 Display Enhancer card and was one of the engineers on the A3000 design team. He enjoys spending time with his family, flying model airplanes, and playing with neat video toys. Contact him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458.*

On Disk

Pass the Word: Interprocess Communication

With the proper ports, your C programs can easily exchange information.

By Eric Giguere

WITH SO MANY people shouting the praises of ARexx, the fact that the Amiga OS itself provides facilities for programs to communicate with each other and Intuition during execution is often forgotten. These *interprocess communication* (IPC) capabilities are vital to the proper functioning of your Amiga.

IPC is accomplished via messages—blocks of data passed from one task to another. To receive these, you must include a message port in your program. (You can create more than one port, but this facility is mostly used by Intuition.) The message port acts as a queueing station for messages: Other programs add messages to the end of the queue, and the system sends a signal (a notification flag) to the task that owns the port. The receiving task then retrieves the messages from the queue and processes the data. When done, the receiver replies to the message so that the sender knows the message was received and processed.

OPEN THE LINES

The straightforward way of creating message ports in C is to use the CreatePort( ) function. Take a look at this example:

```
struct MsgPort *port;
port=CreatePort( "PortName", 0 );
```

The first parameter is the string that represents the name of the new message port. Other tasks use this name to locate the port before sending it messages. The name you use for your message port should be unique and preferably application-specific. Names are case-sensitive, and it is best to use alphanumeric strings only. CreatePort( ) does not allocate new storage for the port name, but merely makes a copy of the pointer to that name. This is fine for statically defined strings, but can be a problem when using arrays. Be careful; placing a new string in the array will change the port name.

The second parameter represents the priority of the port. If two ports do have identical names, the one with the higher priority always takes precedence. Use a zero priority for normal situations.

The CreatePort( ) function returns a pointer to a MsgPort structure, which Exec uses to keep track of a message port's settings. The internals of this structure are not important and should not be altered without just cause. You will need to store the pointer for use with other messaging routines, including the DeletePort( ) function which you use to remove the message port.

Once a task has registered a port with CreatePort( ), other tasks can search for that port using the FindPort( ) function:

```
struct MsgPort *port;
port=FindPort( "PortName" );
```

If FindPort( ) cannot find the desired port, it returns a NULL pointer.

One simple use for FindPort( ) is to ensure the uniqueness of a port name, as in the following example:

```
/* RegisterPort—Like CreatePort, but will not register
* a port if the name is already in use. Returns NULL or a
* pointer to the port. */


struct MsgPort *RegisterPort( UBYTE *name, LONG pri )
  {
    struct MsgPort *port;
    port=NULL;
    Forbid( );
    if( FindPort( name )= =NULL )
      port=CreatePort( name, pri );
    Permit( );
    return( port );
  }
```

Notice the use of the Forbid( ) and Permit( ) functions. Forbid( ) temporarily prevents Exec from switching to another task, while Permit( ) restores multitasking. They are used in the RegisterPort( ) function to ensure that no other task registers a new port in between the calls to FindPort( ) and CreatePort( ). Forbid( ) and Permit( ) should be used with extreme caution and for the shortest possible duration.

"HELLO?"

You send a message to a port with the PutMsg( ) function. First, however, you must declare the message data structure to be passed. Exec puts no restrictions on the format of a message other than it must start with the following structure, which is defined in the exec/ports.h header file:

```
struct Message {
  struct Node  mn_Node;
  struct MsgPort *mn_ReplyPort;
  UWORD  mn_Length;
};
```

The maximum length of a message is 65535 bytes, *including* the size of the Message structure. Store this value in the mn_Length field. The receiving program uses the mn_ReplyPort field to reply to messages; leave it NULL for now. In addition, you should set the mn_Node field's ln_Type subfield to NT_MESSAGE (an Exec-defined constant).

For example, let's define a message structure that adds an ►

array and an integer to the basic Exec Message structure:

```
struct my_message {
   struct Message message;
   char   name[ 20 ];
   int    count;
};
```

To send this message you could use the sequence:

```
struct MsgPort *port;
struct my_message msg;

/* Prepare the message */
msg.message.mn_Node.ln_Type=NT_MESSAGE;
msg.message.mn_Length=sizeof( struct my_message );
msg.message.mn_ReplyPort=NULL; /* no reply expected */

strcpy( msg.name, "Bart" );
msg.count=1;

/* Find the port and send the message */
port=FindPort( "PortName" );
if( port )
   PutMsg( port, (struct Message *) &msg );
```

The PutMsg( ) function takes two parameters: the address of the message port to which to send the message and the address of the message to send. The message will be queued at the message port, and the message port's owner (the receiver) will be notified of its arrival via a signal, if the receiver's signal flag is not already set. Note that a signal only means that at least one message has arrived and is now in the queue at the message port; it doesn't give a count of the number of messages.

Receiving a message is a two-step process. First, the program waits for a message to arrive. The simplest way to do this is with the WaitPort( ) function, which takes a pointer to a message port as its only parameter:

```
WaitPort( port );
```

Your task (which is calling WaitPort( )) will be suspended until a message arrives at the message port. If no messages arrive, it won't ever wake up!

WaitPort( ) reactivates your task when at least one message has arrived or if a message was already waiting. You use the GetMsg( ) function to retrieve each message in the queue, in the order in which each arrived. GetMsg( ) takes a pointer to a message port as a parameter and returns the next message in the port's queue. If there are no more messages left, GetMsg( ) returns a NULL pointer. The example shows GetMsg( ) in action and uses the my_message structure:

```
void ProcessMessages( struct MsgPort *port )
  {
    struct my_message *msg;
    WaitPort( port );
    while( ( msg=(struct my_message *) GetMsg( port ) ) !=NULL )
      printf( "Got message #%d from %s\n", msg->count,
      msg->name );
  }
```

The while loop calls GetMsg( ) until all messages have been retrieved, and the body of the loop contains the code to process the message data. Note that a message's contents are unknown to Exec. If you send messages in one format to a task expecting them to be in another format, strange things can happen. Your tasks must agree on the message protocol (format) before any meaningful communication can occur.

What if your task has two or more message ports? Instead of using the WaitPort( ) function, you can use the Wait( ) function to wait for messages to arrive on more than one port. We'll look at this method later.

CONFIRMATION REQUESTED

What if a task that sent a message wishes to re-use the memory allocated for the message? A sent message is not physically copied into another area of memory, but merely queued into a message port's list. If the sending process is not careful, it may re-use a message before the receiver has a chance to process it, and perhaps corrupt the message port's queue, as well. Reply messages help avoid this.

A reply message is simply a message from the receiver back to the sender, sort of a PutMsg( ) in the reverse direction. Before sending the message to the receiver with PutMsg( ), the sender must allocate its own message port and store a pointer to it in the mn_ReplyPort field of the Message structure. The sender then uses WaitPort( ) to wait for a reply to arrive. The receiver retrieves the message using GetMsg( ). When finished with the message, the receiver sends it back to the port identified in the mn_ReplyPort field with ReplyMsg( ). Note that once the receiver is finished processing a message, it can re-use the message itself to send data (a return value, for example) back to the sender. ReplyMsg( ) takes the pointer to the received message as its only parameter. If the mn_ReplyPort field is NULL, the message will not be returned to the sender; otherwise it will be sent to the specified port. The sender then uses GetMsg( ) to retrieve the reply message and continues with its processing.

Intuition communicates with your tasks by sending messages through the IDCMP facility.

Before your task exits, it should always remove any message ports it created. To do so, call DeletePort( ) with the pointer that was returned by CreatePort( ):

```
DeletePort( port );
```

Note that you should first receive and reply to all messages queued at the port before deleting the port, in case the tasks that sent the messages are waiting for replies.

IDCMP PORTS

Intuition communicates with your tasks by sending messages through the IDCMP (Intuition Direct Communications Message Port) facility. When you open an Intuition window and set IDCMP flags in the NewWindow structure that initializes the window, you are telling Intuition that you want to receive messages for certain events, such as mouse button presses, keyboard input, and so on. To let you receive this input, Intuition automatically opens two message ports—one for your task, and one as a reply port for Intuition. You then use the GetMsg( ) and ReplyMsg( ) functions to receive and reply to Intuition's messages. The UserPort field of the Window structure returned by OpenWindow( ) is the pointer to the message port that should be used to receive Intuition messages. The typical skeleton program for window processing goes as follows:

```
struct Window   *win;
struct IntuiMessage *msg;
win=OpenWindow( ... );
WaitPort( win->UserPort );

while( ( msg=GetMsg( win->UserPort ) ) !=NULL )
  {
    /* do processing here.... then reply */
    ReplyMsg( msg );
  }

CloseWindow( win );
```

Note that the IDCMP message ports are closed automatically by the CloseWindow( ) call.

IDCMP ports are examples of private message ports. The IDCMP ports do not have names and can't be searched for using FindPort( ). They were created using the AddPort( ) function (which I will not describe here). Private ports are rarely used by application programs. If you need more details on IDCMP and private ports, consult the *Amiga ROM Kernel Reference Manual: Libraries & Devices* (Addison-Wesley).

MULTIPLE EXTENSIONS

A task can create several ports and receive messages on all or none of them. The WaitPort( ) function, however, waits for messages to arrive at a single port only. How then do you handle multiple ports? The answer is the Wait( ) function.

Wait( ) accepts a single parameter consisting of a *signal mask*. I said earlier that a signal is a notification flag set whenever one or more messages arrive at a port. A signal is actually one bit within a 32-bit value maintained for each task. A signal bit is allocated to a message port when you create the port. (Note that only 16 bits are available to each task for the program to use, but signal bits can be shared by ports.) You then pass the bitmask of all the signals for which you wish to wait to the Wait( ) function. Your task will be suspended until one of these signal bits gets set.

You generate the bitmask for a message port's signal bit using the following syntax:

```
ULONG bitmask;
bitmask=1L<< port->mp_SigBit;
```

You generate a bitmask for several signals by combining them with OR:

```
bitmask=( 1L<<port1->mp_SigBit ) | ( 1L <<port2->mp_SigBit );
```

Note that you use the *bitwise* OR bar, not the *logical* (double) OR.You pass the bitmask to Wait( ), which will return another bitmask when one or more signal flags have been set. This new bitmask indicates which signals have arrived and hence which message ports you should look at. (Alternatively, you could simply check each message port.)

A task can create several ports and receive messages on all or none of them.

One common situation occurs when your task allocates a public port and also opens an Intuition window. In this case, you wait for messages to arrive at either port, as shown below:

```
ULONG  sig1, sig2, newsigs;
struct MsgPort *port;
struct Window *win;
struct Message *msg;

... /* open port, window, etc. */

sig1=1L<< port->mp_SigBit;
sig2=1L<< win->UserPort->mp_SigBit;

newsigs=Wait( sig1 | sig2 );

if( ( newsigs & sig1 ) !=0 )
   ... /* get messages from "port" */
if( ( newsigs & sig2 ) !=0 )
   ... /* get messages from "win->UserPort" */
```

IPC STEP-BY-STEP

In summary, to send a message, a task should:

1. Create a reply port with CreatePort( ).
2. Initialize the message structure, including the mn_ReplyPort field.
3. Find the receiver's port with FindPort( ).
4. Send the message to the receiver using PutMsg( ).
5. Wait for a reply using WaitPort( ) or Wait( ).
6. Retrieve the reply using GetMsg( ).
7. Before exiting, remove the port using DeletePort( ).

On the receiving end, a task should:

1. Create a public message port with CreatePort( ).
2. Wait for a message to arrive using WaitPort( ) or Wait( ).
3. Process the message.
4. Reply to the message using ReplyMsg( ).
5. Before exiting, remove the port using DeletePort( ).

See the Giguere directory on the accompanying disk for full examples demonstrating these steps. Makefiles are included both for SAS/C 5.10 and Manx C 5.0d, but please check the README file for general directions. The C source files are extensively commented and even include a general-purpose utility function for starting Amiga processes from within a C program. ■

Eric Giguere is the author of the forthcoming Amiga Programmers' Guide to ARexx *and a member of the Computer Systems Group at the University of Waterloo. Write to him c/o* The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or contact him on BIX (giguere) or on Internet (giguere@csg.uwaterloo.ca).*

On Disk

Debugging Memory Errors

By Doug Walker

THEIR INCONSISTENT BEHAVIOR makes memory errors among the most difficult to track down. They may have varying effects on different machines. They may not even occur under a debugger or when you add debugging statements to your program. Compounding the problem is their large volume of related data; most sizable programs make hundreds of allocations, any one of which might be in error.

Take heart! You can harness the power of your Amiga to sift through the reams of allocation/deallocation data to discover where the errors lie. MemLib (in the Walker drawer of the accompanying disk) is a C link library that detects many common memory errors and forces others to show up consistently instead of randomly. Besides enforcing the Ten Commandments of Memory Allocations (see sidebar), the library is designed to:

- Catch as many errors as possible before they cause a crash.
- Force errors to happen on the development machine instead of in the field.
- Compile out completely based on a preprocessor flag.
- Make it easy to determine which allocation is failing.
- Work on *any* Amiga, regardless of configuration.
- Handle AllocMem( )/FreeMem( ) and malloc( )/calloc( )/realloc( )/free( ).
- Allow flexibility in which debugging operations are performed.

MemLib uses the C preprocessor to redirect all calls to AllocMem( ), FreeMem( ), malloc( ), calloc( ), free( ), and realloc( ) to similar functions in MemLib. The library then examines the calls for consistency and correctness and notes the filename and line number of the code allocating or freeing the memory for use in later debugging messages. Figuring out which section of code allocated or freed the problem RAM is one of the hardest parts of debugging memory errors. Memlib uses the C preprocessor symbols __FILE__ and __LINE__ to make it easy to pinpoint where the memory was allocated or freed.

__FILE__ is a string containing the name of the current C source file. If your source file is called myfile.c, it's as if you typed:

```
#define __FILE__ "myfile.c"
```

Similarly, __LINE__ is a constant number that is continually reset to the current line number in the file.

WELL DEFINED

Take a look in memwatch.h (in the Walker directory). If you define the MWDEBUG flag and include memwatch.h in your program, the line

```
#define AllocMem(size,flags) MWAllocMem(size, flags, __FILE__,
    __LINE__)
```

will reroute any calls originally headed for Exec's AllocMem( ) to the MWAllocMem( ) function in the link library. MWAllocMem( ) accepts the same parameters as AllocMem( ), plus __FILE__ and __LINE__. Because __FILE__ and __LINE__ resolve to the current filename and line number, MWAllocMem( ) knows which file and line number allocated the piece of memory. MWAllocMem( ) monitors this information and relays it to you in the event of an error. By the same token, the following memwatch.h define statements replace other common memory allocation and free routines:

```
#define FreeMem(mem,size) MWFreeMem(mem, size, 0, __FILE__,
   __LINE__)
#define malloc(size) MWAllocMem(size, 0, __FILE__,
   __LINE__)
#define calloc(nelt,esize) malloc((nelt)*(esize))
#define free(mem) MWFreeMem(mem, -1, 1, __FILE__,
   __LINE__)
#define realloc(mem,size) MWrealloc(mem,size,__FILE__,
   __LINE__)
```

START DEBUGGING

Adding the MemLib routines to your program is easy. The program-level memory debug routines are controlled by a preprocessor symbol, MWDEBUG. If you do not define the it, the routines are defined to nothing. To link the program-level routines into your code, first define MWDEBUG as:

```
#define MWDEBUG 1
```

in each compilation either in an include file, in the program file, or on the compiler command line. If you do not define MWDEBUG, all the MemWatch routines will disappear, thus adding nothing to your code size. Somewhere after MWDEBUG's #define statement, include memwatch.h in each file that will be allocating or freeing memory. (Don't include mempriv.h; it is for the internal use of the memwatch.lib routines only.) Next, insert a call to the function MWInit( ) in your main( ) program before any memory calls. Finally, place a call to MWTerm( ) in your main( ) program just before it exits. If you use exit( ) to depart from multiple places, try:

```
#define EXIT(rc) {MWTerm( ); exit(rc);}
```

Make your program crash on your machine, instead of on theirs.

and replace all occurrences of exit( ) with EXIT( ). Installing MWTerm( ) with the onexit( ) call also works if you are not using standard output for your debugging output (see below). Finally, recompile all files in your project. If you wish to remove memory debugging later, simply comment out the line that defines MWDEBUG and recompile all files.

The MemLib routines are very compatible with the corresponding system or library routines. MWmalloc( ) is not, however, a complete substitute for the malloc( ) function. While malloc( ) lets you access the last piece of memory that was freed, MWmalloc( ) does not. This "feature" lingers from Unix days when programmers freed lists of memory with techniques such as:

```
list=head;
while(list !=NULL)
{
   free(list);
   list=list->next; /* USING MEMORY AFTER FREE!!! */
}
```

If your program uses free( ) like this, it will not work with MemLib. Otherwise, you should be able to substitute MemLib routines for your normal library routines with no ill effects—except, of course for the bugs MemLib is designed to catch!

ALL ROUTINE

The program-level MemWatch interface is described below. I omitted the allocate and free functions, because they are meant to mimic the system's functions.

```
void MWInit(FILE *debugfile, LONG flags, char *dbfilename);
```

Call MWInit( ) once before any other memory allocation calls. Call it again to reset the location of your debugging output or to change the flags.

The debugfile file is opened with the system's Open( ) function and is used for all debugging messages. If you do not provide a debugfile, MWInit( ) opens the filename specified by dbfilename automatically *when an error occurs* and closes it automatically (with Close( )) when your program exits. If both the debugfile and the dbfilename are NULL, the standard output stream (as returned by the system's Output( ) function) will be used. The intent is for you to provide debugfile if you already have a convenient place for debugging output to go. If you don't, you can pass the name of a console window as the dbfilename:

```
MWInit(NULL, 0, "CON:0/0/639/199/MemLib output");
```

The system will stay out of the way until something interesting happens; then it will open the console window. Finally, if your program runs from the CLI or Shell, you can pass NULL for both debugfile and dbfilename; MWInit( ) will then send output to the Shell window.

The flags parameter can be one or more of the following, combined with OR if necessary:

MWF_NOLOG: If set, the program does not print error or warning messages. Useful only if you want to turn off debugging. Saves some CPU time.

MWF_CHECK: If set, check all allocated memory each time a memory routine is called. Every AllocMem( ), FreeMem( ), malloc( ), or free( ) call causes all allocated RAM to be examined. This can be extremely slow, but it's safe.

MWF_NOFREE: If set, do not free memory still allocated when the program exits. If not set, any memory you set aside and did not free will be released on your behalf.

MWF_NOFTRASH: If set, freed memory will not be altered. This does save some time, but it is valuable to corrupt freed memory to verify you are no longer using it.

MWF_NOFKEEP: If set, free RAM immediately when FreeMem( ) or free( ) is called. If not set, the program keeps "freed" memory on a chain and checks it periodically. If the value changes, you wrote to freed memory. MWF_NOFTRASH implies MWF_NOKEEP, because keeping memory with unknown values is foolish. If this flag is not set, kept memory is freed if the machine runs out of memory.

MWF_NOATRASH: If set, memory will not be corrupted upon allocation. This also saves some time, but it is extremely valuable to alter allocated memory to be sure you aren't relying on side effects for your program to run correctly.

```
void MWTerm(void);
```

Call this routine once after all memory functions have been completed. It will always generate a check of all memory. You must not call any memory allocate/free routines after calling MWTerm( )! You can pass MWTerm( ) to the onexit( ) function if you are not using the AmigaDOS standard output stream Output( ) for your debugging messages. The routine closes Output( ) before calling onexit( ) and requires a valid debug log file.

```
void MWReport(FILE *reportfile, char *title, int level);
```

Call MWReport( ) when you want information on the ►

The Ten Commandments

Avoiding the guru is better than cleaning up after him. To banish him from your program, follow these Ten Commandments of Memory Usage.

I. Thou Shalt Not Use Memory Before Thou Initializest It.

System memory is not guaranteed to be initialized to any particular value, unless you initialize it to zero using the AllocMem( )'s MEMF_CLEAR flag. Because system memory is often all zeros anyway, your program will work in many situations if it assumes the memory will be all zeros. That one time that memory is *not* initialized to zero, your program will go down, and you'll be left sorting through the ashes unable to figure out what happened. MemLib tries to force this error to occur immediately by setting all system memory to the hex value 0xAA (unless you have specified MEMF_CLEAR). Yes, your program will crash, but at least it will crash consistently!

II. Thou Shalt Not Read Memory After Thou Hast Freed It.

In a multitasking system like the Amiga, accessing memory that you have already freed for a read or write is a very bad practice. Again, it may work most of the time; the memory you access may not have been reassigned to another process yet. Even if it has been, it may still contain what you originally put there. MemLib forces the error to occur by filling all freed memory with the hex value 0x55 when you free it. Again, your program will crash, but be glad it does *before* it gets into a user's hands.

III. Thou Shalt Not Write Memory After Thou Hast Freed It.

Disobeying this rule can cause extremely dangerous bugs, because the freed memory you write to may have already been allocated by a new process. MemLib keeps a list of all freed memory. When you call free( ) or FreeMem( ), MemLib does not free the memory, but places it on the freed memory list, instead. You can check the list whenever you want, or MemLib will check it for you when the program exits. If you haven't written to the freed memory, it will contain the junk values that MemLib set. If you did write to it, MemLib detects this and gives you a warning message. (Memlib will free the memory on the freed list if the system gets low on memory.)

IV. Thou Shalt Not Write Past The End Of Thine Allocations.

This is a fairly common error: You didn't allocate a long enough memory buffer and overwrote the end. Perhaps it was a buffer designed to hold a character string, and you forgot the NULL byte terminator. MemLib allocates some extra memory at the beginning and end of each allocation and sets it to a known value. When you ask it to check or when you free the memory, MemLib verifies that the correct values are still there. A change indicates that you overwrote the header or trailer and prompts MemLib to send a warning message.

V. Thou Shalt Not Free Thy Neighbor's Memory.

Sometimes your code will pass a totally fictitious pointer to FreeMem( ) or free( ). This may be related to one of the above problems—using memory before initialization or after freeing, for example. You may be passing a totally random pointer to FreeMem( ), or you may be passing NULL. MemLib has a list of all your

amount of memory your program is using. Reportfile is a file to which to send the report. If reportfile is NULL, MWReport( ) uses the debug log file. Title is a character string that the routine uses to label the dumped output. Set it to NULL for no title. Specifying the amount of detail you want in the report, level takes one of the following values:

MWR_NONE:	Don't print anything.
MWR_SUM:	Print current and total memory usage.
MWR_FULL:	Print a short description of each outstanding allocation.

```
void MWCheck(void);
```

Call this routine when you want to verify all your allocations are clean. If you did not set the MWF_NOCHECK flag, all allocations are checked every time you do an Alloc( ) or Free( ) operation anyway. You might want to use this directly if you set the flag or if you go long periods without allocating memory.

```
void MWLimit(LONG chip, LONG fast);
```

Call this routine if you want to set an artificial cap on the amount of memory available. Any allocations that ask for memory that would push your total above the specified limits will fail, even if memory is available. Keep in mind that this does not take fragmentation into account, and therefore does not guarantee your program will work on a low memory machine! You can simulate out-of-memory conditions by calling MWLimit( ) with (0,0)—no allocations will ever succeed until the limit is raised above the current usage level. The chip and fast parameters set limits on chip and fast RAM, respectively. If the specified limit for a category is −1, the limit will be set at the current usage amount. Thus, any calls to Free( ) will improve your situation. If you want to remove a limit, set it to some extremely large value, such as 0x7fffffff.

LIBRARY SCIENCE

Although you can set many of MemLib's options from the MWInit call, there are some that can only be set by rebuilding the library itself. You can control these features by changing some #defines in mempriv.h.

The define EXTERNAL_LIB enables the use of Commodore's debug.lib and ddebug.lib rather than AmigaDOS file handles for output. It is defined to 0 by default, indicating that debug.lib and ddebug.lib should *not* be used. If you de-

of Memory Allocation

current allocations. Any attempt to free a pointer not on the list results in a warning.

VI. Thou Shalt Free Memory But Once.

This is really a special case of an invalid free value. The memory won't be on the allocated list. Because MemLib also keeps a freed memory list, it can check the freed list and determine if you are freeing the memory twice.

VII. Thou Shalt Not Bear False Witness to FreeMem( ).

FreeMem( ) requires you to tell it the length of the memory to be freed. If the length you specify does not match that which you passed to AllocMem( ), you can "leak" memory to the system and fragment RAM into small chunks. MemLib verifies your size when FreeMem( ) is called. Similarly, it checks the size when AllocMem( ) or malloc( ) is called, and complains about sizes less than or equal to zero.

VIII. Thou Shalt Not Assume Success.

When system memory gets low, you can separate the well-written programs from the hacks very easily. The hacks crash your system. Programs should check all AllocMem( ) and malloc( ) calls for a NULL return value. MemLib allows you to set limits on the amounts of chip and fast RAM you will let your program allocate. There are much better utilities out there now to do this, such as Bill Hawes' Memoration. Whether you use MemLib's limits, Memoration, or some other method, you will also need a "watchdog" program to check for low-memory access as your program attempts to use the NULL pointer returned. If you have an A2500 or A3000, "Enforcer" by Bryce Nesbitt (in the Applications drawer of the accompanying disk) is the tool of choice, by far. Otherwise, get a copy of MemWatch II by John Toebes. It runs on any Amiga and puts up an alert if you write to low memory.

IX. Thou Shalt Givest Back What Thou Art Alloted.

If you fail to free memory that you allocate, nothing bad will happen right away. Eventually, however, the loss may add up, and your system will run out of free memory. If running out of free memory doesn't cause an outright crash, it will certainly prevent you from doing anything interesting until you reboot. MemLib lets you know if your program tries to exit without freeing memory. It will also give you a full report on request of all outstanding allocations, with the filenames and line numbers of the allocators.

X. Thou Shalt Consult The Circular Debugger When Necessary.

The last resort in debugging a difficult problem, the Circular Debugger has brought inspiration to legions of programmers. CDs come in a variety of sizes, and you can usually obtain one with a simple phone call. Look in the telephone directory under "Pizza." Seriously, sometimes you need a break. If a problem gets you down, work on something else for a while, play a game, talk it over with a friend, run around the block, or read a book. You could even try getting some sleep for a change! □

—DW

fine it to 1 instead, MemLib uses the libraries to send its debugging messages out the serial or parallel port. Using debug.lib or ddebug.lib, you can successfully use MemLib on Exec tasks, file systems and handlers, and just about any other Amiga system task or program. With EXTERNAL_LIB defined to 1, MemLib never attempts to open, read, or write AmigaDOS file handles. (See the *Amiga ROM Kernel Reference Manual: Includes & Autodocs* from Addison-Wesley for more information on debug.lib and ddebug.lib.)

The define MW_HEADLEN controls the number of bytes that will be set aside before your allocation to act as a header. Four is the minimum and the default. The header will be checked for trashing when MWCheck( ) is called.

The define MW_HEADSTR sets the value to which header memory will be set. It must be a string at least MW_HEADLEN bytes long.

On the other end, MW_TRAILLEN controls how many bytes will be set aside after your allocation to act as a trailer. The default is 16; you can choose any value for a maximum, but the higher the number, the more memory is wasted each time you call AllocMem( )! The trailer will be checked whenever MWCheck( ) is called.

MW_TRAILSTR specifies the value to which trailer memory will be set. It must be at least a string MW_TRAILLEN bytes long.

MWATRASH sets the value to which newly allocated memory will be set. The default is 0xaa.

Finally, MWFTRASH sets the value to which newly freed memory will be set. The default is 0x55.

MemLib is not a panacea for your memory problems. For one thing, it's big and slow, and not suited to applications pushing the limits of size or speed. For another, it can't catch totally random memory trashes. It can, however, detect many problems before they turn into three-day debugging sessions. I hope it's as good to you as it has been to me. ■

Doug Walker is a founding member of The Software Distillery and currently works for SAS Institute, Inc. on the SAS/C Development System for AmigaDOS. He has worked on the BLink linker, the Amiga Hack icon editor, the NET: network file system, and other projects. Contact him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or on BIX (djwalker), PeopleLink (dwalker), or USENET (walker@unx.sas.com).*

The Off-Line Library

When the routines in your .library files can't give you the answer, turn to your bookshelf.

By Jim Butterfield

KNOWING THE RIGHT answer often is not as important as knowing where to look it up. Years of searching have helped me assemble a core of sources that includes Commodore's official Amiga references and language-specific books.

START WITH DOS

The AmigaDOS Manual ($24.95, Bantam), one of the "Commodore official" documentation books, lets you put the Amiga to work quickly. The heart of "traditional" programming, AmigaDOS lets you access such devices as the keyboard, screen, printer, disk files, and others. The manual is actually three books in one binding: *The AmigaDOS User's Manual*, *The AmigaDOS Developer's Manual*, and *The AmigaDOS Technical Reference Manual*. The first section is an introduction to CLI. Part tutorial, part reference, it uses up a lot of space describing editors ED and EDIT. The heavy-duty reference material follows in the next two sections. The key chapter of the developer's section is "Calling AmigaDOS," which gives a complete list of all DOS library calls: what they do and how to set them up. Using these calls, you can scan directories, open files, handle input and output, and more. The appendix, "Console Input and Output on the Amiga," is also a valuable reference.

The AmigaDOS Technical Reference Manual contains three meaty chapters. "The Filing System" describes how data is organized on disk. It does not cover the FastFileSystem, but if you need to analyze how a floppy disk is put together, this chapter will be your reference. "Amiga Binary File Structure" gives you all the data you need to understand the makeup of a loadable file such as a program, font, or library. It gives details on how such files are set up for memory usage, hunks, and overlay structures. There's also a lot of detail on the structure of object files that are used during program development. Finally, "AmigaDOS Data Structures" describes many of the mechanisms that bring a program into the Amiga and make it a working process. Much of this is described in terms of the BCPL language, which was used internally by the Amiga. All this makes tough reading for the average programmer. If you want to reach advanced features, however, be prepared to wade in and read the details on locks, handles, DOS packets, and device handlers.

The current version of *The AmigaDOS Manual* is for the Workbench 1.2 system, but as programming changes between 1.2 and 1.3 are slight, the information it contains is valid. The user's manual looks a little dated, but the developer's and technical reference sections are still indispensable with 1.3. Version 2.0 of the operating system, however, heralds extensive changes. Available system calls will almost double, new structures will be used, and new techniques will be possible. While properly written "old-style" coding will continue to work, the need for a revised manual is urgent.

Systems with Amiga Basic receive a bonus on the Extras disk. FD files are text files that you can list on screen or print out. For example, with your Extras disk in drive 0, enter: TYPE DF0:FD1.3/EXEC_LIB.FD in the CLI to see a text file detailing calls to the Exec library.

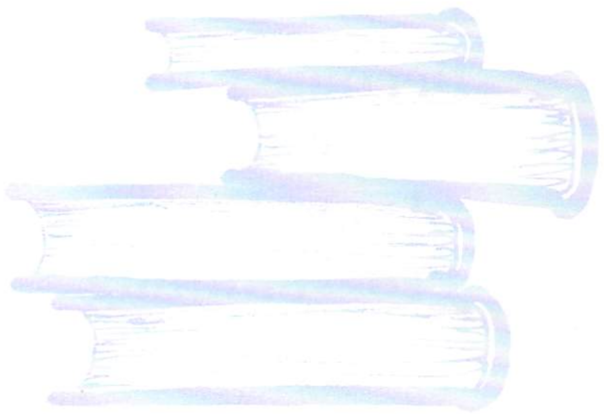
If you prefer a tutorial approach to the operating system, consider ***The Amiga Companion, 2nd Revised Edition*** by Rob Peck ($19.95, IDG Communications) for 1.3 details and the ***AmigaWorld Official AmigaDOS Companion 2.0*** by Bob Ryan ($24.95, IDG Books) for OS 2.0 information. The Monarch Notes of AmigaDOS, Abacus' ***AmigaDOS Quick Reference*** ($9.95) is an inexpensive source for looking up details on CLI/Shell commands or other simple operations. The book's big brother is the popular ***AmigaDOS Inside & Out*** ($19.95), which gives more detailed descriptions and discusses OS 2.0.

BIG BLUE AND FRIENDS

No programmer's bookshelf is complete without Addison-Wesley's three-volume Amiga Technical Reference Series, affectionately called The Big Blue Books for their almost 2000 $8^1/_2 \times 11$ pages. *The AmigaDOS Manual* shows you how to put the computer to work, but these help you add the sizzle. They give you the extra resources to harness the Amiga's graphics, animation, sound, windows, multitasking, and many other features. They also offer a considerable amount of guidance material—suggestions on programming style, cautions on upward compatibility, and a trouble-shooting guide to help debugging.

The ***Amiga Hardware Reference Manual*** ($24.95) gives details on the Amiga's physical organization, including the Amiga custom chips: Agnus, Paula, and Denise. The mechanics of the copper and blitter are described here, together with information on sprites, playfields, audio, and other interfaces. To find out how to harness this hardware, reach for the ***Amiga ROM Kernel Reference Manual: Libraries & Devices*** ($34.95) and the ***Amiga ROM Kernel Reference Manual: Includes & Autodocs*** ($32.95).

Libraries & Devices contains tutorial material on Amiga system functions. The first 13 chapters, which deal with Intuition, have a nice narrative continuity; you can track through the C language examples, which were designed with SAS C. The book then shifts to the Exec and the other libraries; these are good for individual study but do not flow from one subject to another. When the book starts to deal with devices (such as Audio, GamePort, Console, and TrackDisk), it be-



comes harder to read. Keep your AmigaDOS reference book close by. The Big Blue Books deal with technical questions well, but don't give perspective. It may take a while to sort out: which functions are best done entirely from AmigaDOS, which should go to the device, and which call for a mixture of both methods. A copy of ***Inside the Amiga*** by John Thomas Barry ($22.95, Sams/The Waite Group), which covers both topics with plently of SAS C examples, will help, as well.

Includes & AutoDocs is Commodore's official "hard documentation" and gives full detail on calls to resources, devices, and libraries (except AmigaDOS calls). The include files for both C and assembly language are completely listed, as well. If you are looking for offset values for structures or function calls, however, you will find them more quickly in Section H ("Reference Charts"). The book also discusses IFF file structures and the linker libraries, mostly amiga.lib.

If the size (or price) of the Amiga Technical Reference Series is too daunting, there are two alternatives. Rob Peck's popular ***Programmer's Guide to the Amiga*** ($24.95, DATAPATH and Sybex) covers system calls with many C coding examples. Companion disks for C and Modula-2 are also available. Eugene Mortimore's ***Amiga Programmer's Hand-*** ►

Publishers' Addresses

Abacus Software
5370 52nd St. SE
Grand Rapids, MI 49512
616/698-0330

Addison-Wesley Publishing
Route 128
Reading, MA 01867
617/944-3700

Ariadne Ltd.
322 Premier House
10 Greycoat Place
Westminster, London
England SW1P 1SB
071-222-8866

Bantam Books
Bantam Electronic Publishing
666 5th Ave.
New York, NY 10103
800/223-6834, ext. 9479

Commodore Business Machines
Dept. C
1200 Wilson Dr.
West Chester, PA 19380
215/431-9100

Compute! Books
Compute Publications
324 Wendover Ave., Suite 200
Greensboro, NC 27408
919/275-9809

DATAPATH
PO Box 1828
Los Gatos, CA 95031

IDG Books/IDG Communications
80 Elm St.
Peterborough, NH 03458
800/343-0728

Motorola Literature Distribution Center
PO Box 20924
Phoenix, AZ 85036
800/441-2447

Prentice-Hall
Attn: Individual Order Dept.
200 Old Tappan Rd.
Old Tappan, NJ 07675
201/767-5937

Sams/The Waite Group
11711 N. College Ave.
Carmel, IN 46032
317/573-2500

Sassenrath Research
387 N. State St., Suite 200
Ukiah, CA 95482
707/462-4878

Sybex Computer Books
2021 Challenger Dr. #100
Alameda, CA 95401
415/523-8233

TAB Books
Blue Ridge Summit, PA 17294-0850
800/822-8138

book, Vol. I and II ($24.95 each, Sybex) delves into the system calls, libraries, and devices in great detail.

Mapping the Amiga ($22.95, Compute! Books) is different and interesting. Because the operating system pieces move around within RAM, you cannot really map the Amiga, but you can produce mini-maps, or structures showing what these pieces look like wherever they happen to be found. That's what the book does. The three chapters give details of library functions, structures, and hardware registers. The second printing is a good book for looking things up; the first had a few annoying technical mistakes. (To determine which printing your copy is, find the series of numbers in countdown order on the copyright page. If the last number is a 2 or lower, you have the revised edition.)

A more advanced book is ***Guru's Guide (Meditation #1)*** by Carl Sassenrath ($14.95, Sassenrath Research). This one deals with the concepts used in developing the Amiga's multitasking personality.

CHIP DOCS

For official information on the Motorola 68000 chip and its cousins, turn to the manufacturer. You will be glad you did when it comes time to write and debug programs in assembly language. Unfortunately, the documentation situation is a little confusing. Prior to 1990, you could find out everything you needed to know about the 68000 by using the Motorola publication, *MC68000 User's Manual*. Similar volumes were published for the 68020 and 68030. Information on the instruction sets of all chips has now been moved to a new publication, ***M68000 Family Programmer's Reference Manual***. This volume covers the whole range of processor chips, including the 68040, floating point units, and related chips, such as the CPU32, which you may never meet. Even with this new publication, you may still want a copy of the appropriate *User's Manual*—that's where you will find the best description of addressing modes, for example. If you happen upon a pre-1990 manual for the chip of your choice, grab it; the one volume will give you all you need. You might also find that another publication, the ***Family Reference Guide***, gives you a good overview of the whole Motorola chip line. For a complete listing of available documentation, contact Motorola directly.

Keep in mind that data on the 68000 chip doesn't tell you how to get to specific Amiga features, such as the special chips and the calls, to the operating system. You can find this data in *The AmigaDOS Manual* and *Amiga ROM Kernel Reference Manual: Includes & AutoDocs*. Each system call shows the register usage; that's all you need to make the calls work from an assembly language program.

Several books are available (***Amiga Assembly Language Programming*** for $13.95 from TAB Books, to name one) that deal with assembly language programming on the Amiga. Such books show you how to produce simple working programs, but coding is generally written from a single standpoint. For example, some deal only with Intuition calls, and many books deal only with CLI-started programs. I haven't spotted any that give a comprehensive look at the whole Amiga. Investigate before you buy.

MIDDLE C

Most of the Amiga's documentation is written in C, so you should develop at least a browsing capability in this language. Using C in the Amiga environment produces something of a dual-personality situation. Writing code with standard C functions (such as printf, malloc, and fopen) will produce workable programs, but such programs will not tap any of the Amiga's special features. Writing C code that targets the Amiga library calls directly lets you produce efficient and even spectacular programs; however, such programs might not be easily portable to other computers. Many programmers take a middle course, using some standard C and some custom Amiga calls.

Containing both tutorial and reference material, the official guide to C is ***The C Language*** by Kernighan and Ritchie ($32, Prentice-Hall). Look for the revised edition, because C has undergone a number of style changes since the original volume was published in 1978. To develop your Amiga-specific C skills, consult the official Commodore documentation, which is written mostly in C. Keep in mind the style is for the SAS C compiler. If you prefer the Manx Aztec compiler, you will find suggested modifications in the introduction to *Amiga ROM Kernel Reference Manual: Libraries & Devices*.

For a gentler introductory approach to standard C, try the popular ***C Primer Plus*** ($29.95, Sams/The Waite Group).

BASIC READING

The Amiga Basic manual supplied with your machine is fairly good and will get you started. Studying the Amiga Basic demo programs on the Extras disk will reveal some surprisingly advanced techniques that you can steal for your own programs. Abacus' discontinued *Amiga Tricks and Tips* was valuable for the same reasons. The next best (and readily available) book is ***Amiga Graphics Inside and Out*** ($34.95, Abacus), much of which uses Amiga Basic.

LOST LITERATURE

A fine book that is almost lost to this part of the world is *The Kickstart Guide to the Amiga*. It describes the Amiga's internal workings in great detail, contains technical descriptions, and is beautifully written and straightforward. Because the book is no longer available in North America, you have to order it from the British publisher, Ariadne. If you run across a copy at a local computer flea market, grab it. And while you're there, look for back issues of two fine technical magazines which have ceased publication: *The Amiga Transactor* and *The Amigan Apprentice and Journeyman*. Every issue contains solid technical material.

HEY, MR. POSTMAN

Over the past several years, Commodore has been publishing a series of technical notes known as ***Amiga Mail***. Contrary to popular belief, you do not need to be a registered developer to obtain this publication. It contains a lot of good material: sample programs, guidelines for using Amiga features, and extra documentation. Volume I, which covers systems prior to DOS 2.0, is now complete; you can obtain the set for $75. A one-year *Amiga Mail* subscription is available for $45 (add $2.50 per item for shipping to Canada, $5 to other countries) and will supplement your library all year long. ■

Jim Butterfield, a grizzled veteran of the microcomputer wars, has been writing about Commodore machines from the Kim1 to the Amiga. He is also the author of Machine Language for the Commodore 64, 128, and other Commodore Computers. *Write to him c/o* The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458.*

from p. 24

debuggers better, as much because I'm used to them as because of features. Pick the one that best suits your needs; you won't waste your money.

AmigaDOS C Development System 5.10
SAS Institute Inc.
SAS Circle, Box 8000
Cary, NC 27512-8000
919/677-8000
$300
One megabyte recommended.

Aztec C68k/Am-d Developer System 5.0a
Aztec C68k/Am-p Professional System 5.0a
Manx Software Systems
PO Box 55
Shrewsbury, NJ 07702
800/221-0440
$299, $199
One megabyte recommended.

BLITZ BASIC AMOS—The Creator

Blit first, ask questions later.

By Robert D'Asto

"APPLICATIONS, BAH! I bought my Amiga to write *games.*"

Sound like you? Two new BASIC programming implementations, BLITZ BASIC and AMOS—The Creator promise environments groomed to spawn stunning graphics, nimble animations, and abundant aural accompaniment. You'll find everything you need to create dazzling Amiga games with the programming ease of BASIC.

These two BASIC dialects are about as similar to each other as Spanish is to Italian, and both resemble the default standard, Amiga Basic, the way double-chocolate tastes like vanilla. Considering the conventional BASIC norm for graphical razzle-dazzle is usually a few flickering bobs herky-jerkying around on Pixel Flats, such change and improvement in BASIC graphics and sound programming is long overdue.

IF IT MOVES, BLITZ IT

BLITZ BASIC, an Australian import, is a compiled BASIC language utilizing an integrated editor/compiler environment. Besides the language, the non-copy-protected, two-disk BLITZ package includes a number of sample programs with source code, Maestro, a music and sound effects composition utility, and a 134-page, nonindexed manual. The editor's screen and user interface are "Intuition-like" in that pull-down menus, requesters, and a scrolling gadget are employed, but it uses a nonstandard text font, which I found hard to read at times. (The Intuition library is not in the BLITZ disk's libs directory.) Editing features include text-block manipulation, search, and a unique "mousable labels" feature that displays line labels in an area on the right of the screen, allowing quick jumps to various code sections via mouse clicks. The compiler is accessed directly from the editor's menu allowing immediate compilation to memory of the current source code or creation of a stand-alone, executable file on disk.

BLITZ produces applications that are down-to-the-hardware machine code that bypass the Amiga's operating system, thus obviating multitasking and conventional access to the Amiga libraries. I found program exit and return to the Workbench or CLI to be clean, however, allowing full resumption of Amiga system operations. You may distribute your stand-alone executables without license fees, provided you return your registration card.

The BLITZ BASIC language consists of 132 statements and functions, approximately half of which are more or less direct replacements for Amiga Basic keywords. Otherwise the vocabulary is, as you might expect, heavily graphics oriented with commands for direct control of screen configuration, display modes, sprite manipulation, and blitter operations. It also has some unique sound-control statements, such as BEND and VIBRATO. BLITZ directly supports Dual-Playfield, HAM, Extra_HalfBrite, and Double-Buffer modes, and you can quickly set up each with only one or two commands. Overall, I found the language to be more BLITZ than BASIC though, and with emphasis on linear, rather than structured, programming style and terse source code.

Because the Intuition library is nonexistent, BLITZ programs have no windows per se, but they can be approximated. Maestro's interface, which was written in BLITZ BASIC, is a good illustration. Missing too are menu support, random access file operations, and system library access.

The graphic objects used in BLITZ programs are IFF images that you can load with several different statements, depending on how you plan to use the images. As a result, you need an IFF-compatible paint program to create the files; none is supplied.

For fonts, you have the choice of either the default ROM-based type or a user-customized font, which must be drawn with an IFF paint program and is limited to one color and 8 × 8 pixel dimensions for each character. Character strings are of fixed length by default, but you can lengthen them via the DIM statement. Speed of string handling and numeric calculations are in the assembly language fast class, as are the majority of BLITZ operations.

How easy is BLITZ BASIC to use? Herein lies my major criticism. The product's advertisement describes it as "the ideal tool for anyone from beginner to professional to get the Amiga to do graphical gymnastics." A professional? Yes. A beginner? I cannot see someone new to programming or just new to the Amiga with programming experience on another machine getting very far with BLITZ. The manual contains a scant 16-page description of BASIC programming, followed by a seven-page chapter entitled "Simple BLITZ BASIC Concepts" that all too briefly discusses such "simple" topics as dual playfields, HAM, Extra_HalfBrite, blits, hardware sprites, and 8SVX sound samples. This is followed by several terse chapters covering double buffering, vertical blanking intervals, and advanced programming techniques. Reference is made to such concepts as bitplanes, DMA, and Copper programming, while providing no definitions for these terms. The majority of the documentation is devoted to discussions of the individual BLITZ statements, but the descriptions given for some of the specialized BLITZ keywords will frustrate novices with their brevity. For example, the description of the BMODE command states it "will set the 'blitter nasty' mode. For more information. . .see the *Amiga Hardware Reference Manual.*" I doubt many beginners own a copy of this highly technical book. There is also no instruction whatever provided for the use of HAM mode screens. There *are* impressive demo programs on disk with source code provided for study, but their documentation comments are sparse and not very explanatory.

Tapping the Amiga's sound and graphics prowess, however, is BLITZ BASIC'S forte, and in this regard, its abilities quickly become apparent. A benchmark analysis of BLITZ versus

Amiga Basic would be no contest. BLITZ graphics and animated objects fly through display configurations unknown to conventional Amiga Basic applications with machine code snap. When properly set up, scrolls and sprite animation are also smooth and flickerless, and can be created with surprisingly compact source code.

AMOS UNUSUAL BASIC

From England, AMOS—The Creator (aka AMOS Basic) consists of six non-copy-protected disks and an almost 300-page manual. AMOS is an interpreted BASIC incorporating an integrated editor as Amiga Basic does. The similarity stops there, however; AMOS is much faster and more feature-laden than Amiga Basic. A public-domain runtime version of the interpreter, RAMOS, is supplied, so you can distribute AMOS applications, as long as you provide the appropriate acknowledgements. The enclosed literature also promises a compiler, AREXX interface, and 3-D graphics accessory package in the near future.

The AMOS Basic system is undeniably huge, consisting of over 500 commands and functions and including a specialized sublanguage, AMAL, for high-speed animation effects. Supported display modes include HAM, Extra_HalfBrite, Dual-Playfield, and Double-Buffer, plus special graphics effects such as fading, flashing, rainbow palettes, blits, Copper list control, text masks, and others. You can create graphic objects with the supplied sprite editor and animate them as sprites or bobs with AMAL, which executes independent of the host AMOS program. Each AMAL program controls the movement of a single screen object, and you can use interrupts to execute multiple AMAL programs simultaneously. AMOS also provides numerous menu design options, including movable and pop-up menus, as well as keyboard shortcuts, control of text color and font, and embedded graphics. Sound control ranges from special effects keywords, such as BOOM and SHOOT for the ubiquitous explosion and gunshot noises, to more complex music, waveform, and sound envelope control commands. AMOS programs can also load and play sound sample files and SMUS music files converted to a special format with a supplied utility.

Unlike BLITZ, AMOS provides extensive file-handling commands and functions for both sequential and random-access files, as well as support of the CrossDOS system from Consultron, allowing access to PC and ST format files from within AMOS programs. This makes AMOS suitable for designing database applications and file manipulation utilities, not just games. Also in contrast to BLITZ, AMOS lets you access the Amiga Exec, graphics, DOS, and Intuition libraries. There are no .bmap files (as in Amiga Basic), however, to set up the CPU registers for calling library routines; you must do this first by directly manipulating the registers via special AMOS functions.

AMOS Basic is impressive in scope and command, with the most extensive, high-level control of Amiga graphics, sound, and animation I have seen in any BASIC implementation, but it does have a down side. Regardless of previous BASIC experience, learning the ins and outs of the AMOS system requires time and dedication. Additionally, when starting up the AMOS editor you are greeted by a screen with blinking cursor and editor options displayed in gadgets at its top, rather than conventional pull-down menus. Overall, I found this and other nonstandard interface features to be an unwelcome distraction. The editor's functions are, however, ample and well documented, and it did not take me long to adapt. Programming with AMOS is also done in a manner similar to Amiga Basic. Source code is either written with the editor or loaded from disk and then run with a "menu" selection. Source code errors cause the program to abort, and the offending line is indicated in the editor for correction in usual interpreter fashion. There is also an immediate mode, allowing instant execution of code fragments. AMOS programs conform to multitasking protocols and do run very fast, with smooth, no-flicker animation and scrolling effects. I see nothing to prevent a programmer from creating commercial-class applications with AMOS Basic.

THE BLITTER END

In the graphics and sound departments, I found little to disappoint in BLITZ BASIC and AMOS. Both appear capable of creating games and audio/visual applications rivaling those produced with any Amiga language, and with substantially less development time. As a structured BASIC language, AMOS does contain familiar elements, particularly in the area of control structures. Otherwise it was unfamiliar and demanded considerable study before I became adept. The AMOS approach, however, is more high-level than BLITZ, requiring far less prior knowledge of Amiga-specific hardware and programming concepts, and providing superior documentation. In contrast, I found BLITZ BASIC'S compact vocabulary and Amiga-style interface more conducive to writing quick, getting-the-hang-of-it applications, and therefore more immediately appealing. Unfortunately, the sparse documentation will leave novice Amiga programmers in the dark.

BASIC is an acronym for Beginner's All-purpose Symbolic Instruction Code. I see BLITZ as neither a language for beginners nor all-purpose, but, for savvy Amiga programmers, it does allow masterful control of Amiga graphics and sound, as well as machine code speed. AMOS does qualify as all-purpose and is closer to a beginner's language, but it may not be suitable for the already experienced Amiga programmer who does not wish to invest the time necessary to learn a new and extensive language system. If you have done a little BASIC programming and if writing graphics and sound-intensive applications is your fondest desire, AMOS has much to offer. If you have done a lot of Amiga programming and have grown tired of the tedious coding requirements of game and AV applications and hanker for a quick, BASIC-like system to fling and sing your artwork through Amiga time and space, BLITZ BASIC may be just the ticket. ■

BLITZ BASIC
M.A.S.T. Memory and Storage Technology, Inc.
1395 Greg St.
Sparks, NV 89431
702/359-0444
$149
No special requirements.

AMOS—The Creator
Mandarin Software
distributed by American Software Distributors
502 E. Anthony Dr.
Urbana, IL 61801
217/384-2050
$99.95
No special requirements.

LETTERS

Flames, suggestions, and cheers from readers.

LOOKING FOR LESSONS

Can you recommend someplace where I might find someone to teach me machine language on the Amiga? If you do not know of a place where I can receive "human" help, could you recommend some books? I already have Abacus' *Amiga Machine Language* and *Advanced System Programmer's Guide for the Amiga.*

Ryan Fleming
Westville, Indiana

While we couldn't find any organized Amiga programming classes in your area, you might try contacting the Amiga Users of Michiana (2609 Mishawaka Ave., South Bend, IN 46615, 219/287-3344), Logansport Commodore Club (219/223-4542), or Fox Valley Amuse (PO Box 4125, Aurora, IL 60507-4125, 708/759-1590). Perhaps a member of one of these users' groups could tutor you. For book ideas, consult "The Off-Line Library" on p. 42.

HONK IF YOU LOVE AMIGAS

I hope the *Tech Journal* will open some useful links with the best Amiga engineers—wherever they happen to be located! It is obvious that there is a great deal of interest in the Amiga in the UK and Europe. Please don't overlook better ideas just because they "weren't invented here."

Somehow, it should become a consistent goal to establish an outstanding relationship between Amiga engineers and technical users to develop a level of pride and elan beyond that of the Apple users with their little colored decals for car windows! I believe that this is a very important aspect of the particular care we Amiga users have taken in choosing our machines and the amount of time we invest in improving our understanding of the elegant, and sometimes less than excellent, details of their physical and software systems.

I'm proud I chose the Amiga. Just like Hewlett-Packard hand calculators, each new model is going to be even better. Amigas are much more exciting because so many good ideas come from users.

Ralph Marler
Portsmouth, New Hampshire

LET'S HEAR IT

We're ready to listen. Just write to Letters to the Editor, The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or speak your mind in* The AmigaWorld Tech Journal *conference on BIX. Letters and messages may be edited for space and clarity.* ■

Who Helps Amiga Pros?

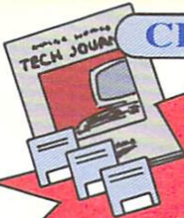
BIX -- the online service for people who know Amiga.

- ❑ Connect with the Commodore technical team
- ❑ Get quick answers to tough coding questions
- ❑ Interact with other Amiga Developers
- ❑ Download source code, utilities and other programs
- ❑ Keep up with the latest Amiga developments
- ❑ Send and receive private e-mail with binary attachments
- ❑ Chat with other Amiga pros in real time

All for just $39 for three months plus $3 per connect hour weeknights and weekends or $6 per connect hour weekdays via BT Tymnet.

Don't miss out! Just have your computer and modem call 800-225-4129 or 617-861-9767 and subscribe on-line. It's easy, at the Login prompt enter bix and at the Name? prompt enter bix.amiga.

BIX 800-227-2983 or 603-924-7681



CHARTER SUBSCRIPTION OFFER

SUBSCRIBE & SAVE

AVAILABLE ONLY FROM AMIGA WORLD

THE AMIGAWORLD

TECH JOURNAL

☑ **YES!** enter my one-year (6 bi-monthly issues, plus 6 invaluable disks) Charter Subscription to The AmigaWorld Tech Journal for the special price of $59.95. That's a savings of $35.75 off the single copy price. If at any time I'm not satisfied with The AmigaWorld Tech Journal, I'm entitled to receive a full refund — no questions asked.

Canada and Mexico $74.95. Foreign Surface $84.95, Foreign Airmail $99.95. Payment required in U.S. funds drawn on U.S. bank. Available March 15.

Name ____________________

Address ____________________

City ____________ State ______ Zip ______

TS41I

☐ Check/Money order enclosed

Charge my: ☐ Mastercard ☐ AMEX ☐ Visa ☐ Discover

Account# ____________ Exp. ______

Signature ____________________

Or call 800-343-0728 ☎ 603-924-0100 for immediate service.

NEW! 6 BI-MONTHLY ISSUES PLUS 6 INVALUABLE DISKS

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT #73 PETERBOROUGH, NH

POSTAGE WILL BE PAID BY ADDRESSEE

The AmigaWorld Tech Journal
P.O. Box 802
Peterborough, NH 03458-9971

The AmigaWorld

TECH JOURNAL

CHARTER OFFER!

☑ **YES!** Enter my one-year (6 bi-monthly issues, plus 6 invaluable disks) Charter Subscription to The AmigaWorld Tech Journal for the special price of $59.95. **That's a savings of $35.75** off the single copy price. If at any time I'm not satisfied with The AmigaWorld Tech Journal, I'm entitled to receive a full refund — no questions asked.

Name ____________________

Address ____________________

City ____________ State ______ Zip ______

TL41I

☐ Check /Money order enclosed

Charge my: ☐ MasterCard ☐ Visa ☐ Amex ☐ Discover

Account# ____________ Exp ______

Signature ____________________

Canada and Mexico $74.95. Foreign Surface $84.95, Foreign Airmail $99.95. Payment required in U.S. Funds drawn on U.S. Bank. Available March 15.

Or call 800-343-0728 ☎ 603-924-0100 for immediate service.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED
STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 73 PETERBOROUGH, NH

POSTAGE WILL BE PAID BY ADDRESSEE

AmigaWorld Tech Journal
P.O. Box 802
Peterborough, NH 03458-9971

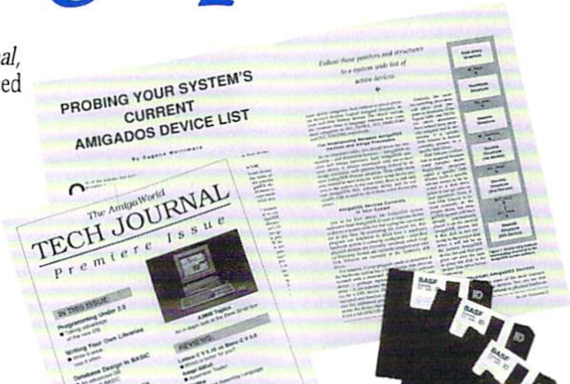
CHARTER OFFER

A source of technical information for the serious Amiga professional.

Introducing *The AmigaWorld Tech Journal,* the new source to turn to for the advanced technical information you crave.

Whether you're a programmer or a developer of software or hardware, you simply can't find a more useful publication than this. Each big, bi-monthly issue is packed with fresh, authoritative strategies and advice to help you fuel the power of your computing.

Trying to get better results from your BASIC compiler? Looking for good Public Domain programming tools on the networks and bulletin boards? Like to keep current on Commodore's new standards? Want to dig deeper into your operating system and even write your own libraries? Then *The AmigaWorld Tech Journal* is for you!


PROBING YOUR SYSTEM'S CURRENT AMIGADOS DEVICE LIST
The AmigaWorld TECH JOURNAL
Premiere Issue

Every issue includes a valuable companion disk!

Our authors are programmers themselves, seasoned professionals who rank among the Amiga community's foremost experts. You'll benefit from their knowledge and insight on C, BASIC, Assembly, Modula-2, ARexx and the operating system—in addition to advanced video, MIDI, speech and lots more.

Sure, other programming publications may include *some* technical information, but none devote *every single page* to heavyweight techniques, hard-core tutorials, invaluable reviews, listings and utilities as we do.

And only *The AmigaWorld Tech Journal* boasts of a technical advisory board comprised of industry peers. Indeed, our articles undergo a scrupulous editing and screening process. So you can rest assured our contents are not only accurate, but completely up-to-date as well.

Plus! Each issue comes with a valuable companion disk, including executable code, source code and the required libraries for all our program examples—plus the recommended PD utilities, demos of new commercial tools and other helpful surprises. These disks will save you the time, money and hassle of downloading PD utilities, typing in exhaustive listings, tracking down errors or making phone calls to on-line networks.

The AmigaWorld
TECH JOURNAL

In every issue of *The AmigaWorld Tech Journal,* you'll find...

- Practical hardware and software reviews, in- cluding detailed comparisons, benchmark results and specs
- Step-by-step, high-end tutorials on such topics as porting your work to 2.0, debugging, using SMPTE time code, etc.
- The latest in graphics programming, featuring algorithms and techniques for texture mapping, hidden-line removal and more
- TNT (tips, news and tools), a column covering commercial software, books and talk on the networks
- Programming utilities from PD disks, bulletin board systems and networks
- Wise buys in new products—from language system upgrades to accelerator boards to edit- ing systems and more.

The fact is, there's no other publication like *The AmigaWorld Tech Journal* available. It's all the tips and techniques you need. All in one single source. So subscribe now and get the most out of your Amiga programming. Get six fact-filled issues. And six jam-packed disks. All at special Charter savings. Call 1-800-343-0728 or complete and return the savings form below—today!

To order, use this handy savings form.

Charter Savings Form

☑ *Yes!* Enter my one-year (6 issues, *plus* 6 invaluable disks) Charter Subscription to *The AmigaWorld Tech Journal* for just $59.95. That's a special savings of $29.75 off the single-copy price. If at any time I'm not satisfied with *The AmigaWorld Tech Journal,* I'm entitled to receive a full refund—no questions asked!

Name ____________________

Address ____________________

City __________ State ______ Zip ______

☐ Check or money order enclosed. ☐ Charge my:
☐ MasterCard ☐ Visa ☐ Discover ☐ American Express

Account No. __________ Exp. Date ______

Signature ____________________

Satisfaction Guaranteed!
Or your money back!

Canada and Mexico, $74.95.
Foreign surface, $84.97.
Foreign airmail, $99.95.
Payment required in U.S. funds drawn on U.S. bank.

Complete and mail to:
The AmigaWorld Tech Journal
P.O. Box 802, 80 Elm Street
Peterborough, NH 03458
T691

For faster service, call toll-free 1-800-343-0728

TAKE THIS JOB AND LOVE IT!

The fastest growing video technology company in the world is looking for technical support specialists. We're looking for the kind of person that enjoys solving problems. If you have the unique ability to; hear a problem, ask the right questions, analyze the answers, and then provide a perfect solution, you belong with NewTek. You'll be a key member of the best support team in the business. Best of all, you'll be working for a company that has an intense commitment to serving its customers.

We're looking for rare individuals with as many of the following qualities as possible:

- Excellent written and verbal communication skills
- Experience in video production and editing
- Knowledge of 3D animation and computer graphics
- Knowledge of Amiga software and peripherals
- Experience with Digi-View and Digi-Paint
- Experience with the Video Toaster
- Work experience in the technical support field

The Video Toaster is changing the way the world thinks about video production. The Toaster is being used everywhere from networks to cable stations and our users range from rock stars to wedding videographers. Helping producers, artists, and video makers use this powerful tool to revolutionize video is an exciting and rewarding career. If you're ready to make a change for the better call Kiki Stockhammer at 913-354-1146.

215 S.E. Eighth St.
Topeka, KS 66603
913-354-1146
FAX: 913-354-1584

NEWTEK
INCORPORATED